Р. ЭКХАУЗ

Л. МОРРИС

МИНИ ЭВМ:

ОРГАНИЗАЦИЯ И ПРОГРАММИРОВАНИЕ

МИНИ-ЭВМ: ОРГАНИЗАЦИЯ И ПРОГРАММИРОВАНИЕ

MINICOMPUTER SYSTEMS

ORGANIZATION, PROGRAMMING, AND APPLICATIONS (PDP-11)

RICHARD H. ECKHOUSE, JR.
Research and Development Group,
Digital Equipment Corporation,
Maynard, Massachusetts

L. ROBERT MORRIS
Department of Systems Engineering
and Computing Science,
Carleton University,
Ottawa, Ontario

PRENTICE-HALL, INC ENGLEWOOD CLIFFS, NEW JERSEY

Р. ЭКХАУЗ, Л. МОРРИС

МИНИ-ЭВМ: ОРГАНИЗАЦИЯ И ПРОГРАММИРОВАНИЕ

Перевод с английского
А. Ф. КОНДРАТЮКА
и Л. С. ЧЕРНЯКА

Дополнение к русскому изданию
Г. П. ВАСИЛЬЕВА и Л. С. ЧЕРНЯКА

Под редакцией и с предисловием
Г. П. ВАСИЛЬЕВА

МОСКВА „ФИНАНСЫ И СТАТИСТИКА" 1983

ББК 32.973
Э44

Экхауз Р., Моррис Л.

Э44 Мини-ЭВМ: Организация и программирование/Пер. с англ. А. Ф. Кондратюка и Л. С. Черняка; Дополнение к русскому изданию Г. П. Васильева и Л. С. Черняка; Под ред. и с предисловием Г. П. Васильева.— М.: Финансы и статистика, 1983. — 359 с., ил.

В пер.: 1 р. 80 к.

В книге изложены основные идеи по архитектуре мини-ЭВМ, структуре программного обеспечения, организации, программированию и применению вычислительных систем в целом. Содержатся программы и схемы, поясняющие работу оборудования, а также системы в целом. Предлагается готовая базовая операционная система. Показана система ввода-вывода, обработки прерываний и распределения ресурсов.

Для всех категорий пользователей мини-ЭВМ, системных программистов, разработчиков систем, инженеров по электронной аппаратуре, аспирантов и студентов.

Э $\frac{2405000000-145}{010\,(01)-83}$ 132—83

ББК 32.973
6Ф7.3

© 1979, Prentice-Hall, Inc., Englewood Cliffs, N. J.
© Перевод на русский язык, предисловие, примечания, дополнение к переводу „Финансы и статистика“, 1983

ПРЕДИСЛОВИЕ К РУССКОМУ ПЕРЕВОДУ

Предлагаемый советскому читателю перевод второго издания книги Р. Экхауза и Л. Морриса, по сути дела, представляет собой введение в организацию малых вычислительных систем и программирование на мини-ЭВМ. В ней рассматриваются фундаментальные идеи, на которых базируются архитектура и структура мини-машин, а также то, каким образом логическая организация ЭВМ отражается на программировании и методах работы с данными. Рассмотрение вопросов, связанных с функционированием и использованием мини-ЭВМ, на примере машин семейства PDP-11, во-первых, позволяет конкретизировать излагаемый материал, а во-вторых, полезно само по себе, поскольку речь идет о мини-ЭВМ, едва ли не самых распространенных в мире. Книга включает информацию об общей организации аппаратных средств системы PDP-11, знакомство с которой будет способствовать наиболее эффективному использованию аппаратных возможностей при программировании на языке MACRO-11. Более того, изложение материала сопровождается значительным количеством примеров программирования.

Все это делает книгу полезной как программистам, пишущим прикладные программы на языке MACRO-11, так и тем, кто использует лишь стандартные возможности системы. Наиболее тщательно материал должен быть проработан системными программистами, участвующими в создании новых операционных систем и управляющих программ. Это, естественно, не означает, что каждым из них книга должна быть проштудирована от первой до последней страницы, прежде чем он приступит к практическому программированию. В зависимости от характера работы тому или иному программисту могут понадобиться лишь отдельные главы.

Структура и содержание монографии, а также характер изложения позволяют с успехом использовать ее в качестве учебного пособия по различным курсам, в том числе и вузовским.

При переводе ставилась цель сделать книгу более доступной для широкого круга читателей. Поэтому из оригинала были исключены некоторые разделы, не соответствующие, по мнению переводчиков, общей структуре монографии Кроме того, учитывая некоторую схематичность рассмотрения вопросов, связанных с описанием характеристик конкретных периферийных устройств комплексов PDP-11 и их операционных систем, к переводу сделано дополнение, в котором приведены общие сведения о стандартных технических и программных средствах.

Будучи одними из наиболее распространенных, мини-ЭВМ PDP-11 отражают основные тенденции весьма важного направления развития вычислительной техники. Поэтому помимо чисто утилитарного значения материал книги в известной мере позволяет критически осмыслить тенденции и проблемы вычислительной техники в целом.

В заключение следует сказать, что в связи с массовым производством мини-машин семейства СМ ЭВМ предлагаемая книга может представить значительный интерес для широкого круга специалистов.

Г. П. Васильев

ПРЕДИСЛОВИЕ

Есть некое очарование, которое испытывает каждый, применяя небольшую отдельную ЭВМ. Оно рождается ощущением полной власти над целой вычислительной системой и сознанием того, что все можно попробовать и все можно сделать. Пионеры ЭВМ, создавая и применяя первые машины, почувствовали это очарование, оно передалось и современным системным программистам.

Некоторые специалисты могут возразить, что описанное ощущение сродни ностальгии и неуместно в замкнутом мире машинных операций. Они подчеркивают, что обычный средний пользователь совершенно удовлетворен программированием на языке высокого уровня и счастлив, что ему удается игнорировать большинство аспектов оборудования, программного обеспечения и создания системы в целом.

Эта книга предназначена не для тех, кто так думает. Она для тех, кто, будучи очарован машиной, желает понять магику, окружающую применяемые устройства. Такие пользователи (системные программисты, разработчики систем, ученые, инженеры, специалисты по прикладным вопросам) хотят знать несколько больше об организации ЭВМ, программировании на уровне ассемблера. Именно в расчете на них и создавалась эта книга.

Таким образом, книга «Мини-ЭВМ» — это введение в *программирование*, *организацию* и *применение* малых вычислительных систем. Как и в первой редакции, центральной темой ее являются фундаментальные идеи *архитектуры* и *структуры* ЭВМ, оборудования и программного обеспечения, а также использование этих концепций в *разработке программ*, *собирающих данные* и *производящих манипуляции* над ними. Во второй редакции такие темы были усилены и расширены с целью облегчения проверки взаимодействия алгоритмов, программ и структур данных при разработке эффективного программного обеспечения. Включен новый материал, который имеет отношение к рассматриваемому предмету. Это этапы обработки машинных команд (со ссылками на применение их на микропрограммируемых машинах); применение расширенных вычислений для чисел с фиксированной и плавающей точкой; рассмотрение концепций иерархии памяти и семейства машин. Усилено внимание к таким системным программам, как ассемблеры, редакторы, связывающие загрузчики и другие, помогающие эффективно разрабатывать модульный код. Рассматривается несколько аспектов совместного использования оборудования и программного обеспечения на примерах программ

ввода-вывода с использованием прерываний и прямого доступа к памяти. Объясняются роль и функции операционных систем, а также документации, которой должна располагать современная многопрограммная система. Важной новой чертой расширенного варианта книги является то, что рассмотренные концепции подкреплены и пояснены многочисленными примерами программ, ориентированных на работу *оборудования* (аналого-цифровых преобразователей, интерактивных графических процессоров) и реализующих *алгоритмы* (автокорреляции, быстрого преобразования Фурье), применяемые при *обработке научных данных*. В частности, включено описание программ для осциллоскопа, а также обсуждается вопрос улучшения программного обеспечения путем внедрения микропрограммирования. Эти добавления излагаются в духе первого издания, а именно путем применения программ на уровне языка Ассемблера. Такой подход позволяет легче объяснить идеи и концепции сделанных добавлений. Наша философия проста: детальное знакомство в конечном счете приводит к пониманию.

Вторая редакция настоящей книги появилась в связи с необходимостью обучить организации вычислений и программированию на языке Ассемблера группы студентов, широко использующих разные ЭВМ. Во всех случаях выделялись фундаментальные концепции оборудования и программного обеспечения ЭВМ, особенно акцентировалось их воздействие на более модные вычислительные системы и их организацию. После испытания различных средств для решения этих задач было признано, что машина реального времени является более предпочтительной для иллюстрации изложенных выше концепций по сравнению с моделирующей машиной.

Среди машин реального времени наиболее распространены и дешевы мини-ЭВМ. Кроме того, они характеризуются такими свойствами, которыми раньше обладали только большие системы (например, совмещение операций, прерывания, наличие наборов регистров, встроенное оборудование для умножения/деления и и т. д.). Эти малые машины входят как составные части во многие системы. Учитывая вышесказанное, мы выбрали именно их для иллюстрации организации и программирования на уровне языка Ассемблера.

Предполагается, что читатель знаком с основами процедурно-ориентированных языков. Поскольку FORTRAN наиболее распространен и изучен, он был выбран для иллюстрации многих примеров. Однако выбор языка высокого уровня для иллюстрации идей, продемонстрированных уже на более низком уровне (MACRO-11), служит не только для простого подтверждения правильности этой идеи. Принципы и способы программирования в общем не зависят от языка, следовательно, большинство языков различных уровней может быть использовано для иллюстрации новых идей. Поэтому читатель этой книги наряду с изучением фундаментальных концепций цифровых ЭВМ и программирования

будет постигать конкретные детали программирования для PDP-11.

Курс программирования на машинном языке и языке Ассемблера, как правило, является уникальным для конкретной машины. Часто такой курс является единственным средством для знакомства и получения некоторого опыта работы с реальной архитектурой ЭВМ. Последующие курсы обучения обычно посвящены логическим (в противоположность физическим) машинам, управляемым операционными системами, отражающими структуру, а значит, и поведение базовой машины. Студента знакомят с основными характеристиками физического уровня машины, манипулируя которыми можно создать довольно сложную логическую ЭВМ. Наша же цель дать столько материала «вглубь и вширь», сколько сможем.

Выбор PDP-11 в качестве реальной машины объясняется несколькими причинами. В ряду таковых можно выделить широкую популярность этой машины и большое число реализованных в ней различных возможностей: богатое множество команд, гибкость формирования адресов с использованием регистров, легкость манипуляций устройствами ввода-вывода. Последующий рост и развитие числа моделей семейства PDP-11, разнообразие действующих периферийных устройств, множество операционных систем сделали машины этого семейства уникальными среди других средств вычислительной техники. Достоинства архитектуры PDP-11 подтверждаются не только популярностью машин среди пользователей, но также исследованиями в этой области. Поэтому значительное число свойств PDP-11 было «эмулировано» в процессорах, созданных другими фирмами, и сама Digital Equipment Corporation (DEC) выпустила свою расширенную, 11-ю модель VAX-11. Некоторые особенности этой модели описываются в данном издании.

В общем, мы старались сформулировать некоторое понятие или идею, а затем проиллюстрировать ее воплощение на PDP-11. Поскольку в данной книге не рассматриваются полностью такие вопросы, как структуры данных или архитектура машины, мы будем часто развивать только принципы, касающиеся этих тем, чтобы в дальнейшем можно было с помощью программ проиллюстрировать соответствующие понятия. Мы выражаем надежду, что те, у кого нет машины или она другого типа, будут в состоянии понять представленные на рассмотрение концепции и соотнести их с собственным опытом. Представим теперь основное содержание книги.

Глава 1 — это краткое введение в те области науки и техники, которые охватило наступление ЭВМ.

Глава 2 открывается обсуждением основных принципов ЭВМ, совершенно не зависящих от специфики какой-либо конкретной машины. Здесь рассматриваются вычислительные машины и их организация. При изложении вопросов программного обеспечения основное внимание уделено типам команд, разработке программ

и их блок-схем, кодированию и символическому кодированию. Затем рассматриваются адресация операндов и методы задания адресов, общие для большинства вычислительных машин. Показаны такие основные этапы обработки команд, как реализация функций символического Ассемблера — программы, представляющей собой средство программирования.

Глава 3 посвящена организации и структуре PDP-11. Сюда включены представление данных, форматы команд, режимы адресации. На простых примерах показано, как ЭВМ выполняет основные операции. Затем расширенные атрибуты ассемблеров (например, директивы) вводятся в язык MACRO-11. Обсуждается понятие архитектуры машины и организации памяти, а также влияние организации памяти на эффективность выполнения программ. Поясняется идея устройства расширения памяти и управления ею, а также защиты программ.

Глава 4 описывает более сложные способы программирования и структуры программ. Объясняются понятия организации программы и модульного программирования, включая подпрограммы, сопрограммы и макрокоманды. Приводятся примеры. Разбирается написание программ в позиционно-независимом коде, реентерабельность и рекурсия. Для улучшения понимания изложение подкрепляется многочисленными примерами.

Глава 5 представляет собой введение в структуры данных и манипуляции ими. Приводятся способы манипуляции байтами и выполнения логических операций, основные алгоритмы арифметики умножения/деления целых чисел и чисел повышенной точности, а также арифметики плавающей точки. Преимущества встроенной аппаратуры для выполнения расширенных арифметических операций в PDP-11 иллюстрируются рядом примеров. Объясняются отношения между внутренними и внешними формами данных. Приводимые алгоритмы часто являются самонастраивающимися. Таким образом, глава содержит довольно обширное описание способов представления, хранения и манипуляции структурами данных, такими, как массивы, стеки, очереди, списки. В главу включено также описание хеширования и создания пакетов. Приводятся многочисленные примеры, иллюстрирующие взаимосвязь между архитектурой ЭВМ (особенно способами адресации) и программным обеспечением обработки различных структур данных. Для демонстрации этих понятий используется символический Ассемблер.

Глава 6 посвящена архитектуре и организации ЭВМ на микромашинном уровне. Описываются самые важные блоки ЭВМ, общие регистры и арифметическое устройство. Показано, как они могут быть связаны друг с другом и управляться с помощью программ. Это продемонстрировано на конкретном примере использования шины данных PDP-11/60. Детально изложены основные ресурсы микромашинного уровня PDP-11/60. Описаны возможности управления этими элементами с помощью микропрограммы, выполняющей как существующие, так и новые коман-

ды машинного языка PDP-11, а также управляющей последовательностью выбора самих команд PDP-11. PDP-11/60 — реальная машина, предназначенная для микропрограммирования. Ее микроструктура проста для понимания, однако достаточно мощная, чтобы реализовать и продемонстрировать такие идеи, как параллельная обработка.

Само по себе наличие программы для ЭВМ мало что значит, если нет возможности обеспечить ее данными и затем вывести результаты. Поэтому *глава 7* включает описание программирования ввода-вывода вплоть до работы с устройствами, управляемыми по прерываниям. В ней описывается также программное обеспечение, которое обычно поставляется разработчиком и имеет целью дать пользователю возможность эффективно применять устройства ввода-вывода и помочь ему в написании соответствующих программ. Приведенные примеры основаны на использовании реальной системы (в данном случае RT-11).

Глава 8 представляет собой краткое описание некоторых других ресурсов системного программного обеспечения, которые можно встретить на большей части малых ЭВМ. Обсуждаются функции редактора (включая специальные их версии, ориентированные на видеотерминалы и называемые экранными (window) редакторами), процессы связывания и загрузки и возможности динамического отладчика, работающего в интерактивном режиме.

Далее глава 8 делится на две более или менее независимые части. Первая продолжает развивать тематику, представляющую интерес с точки зрения применения мини-машин, а во второй на более высоком уровне иллюстрируется тематика предыдущих глав.

В *главе 9* освещается значение и использование дискретного и быстрого преобразований Фурье (DFT/FFT) и на примере их раскрываются и поясняются такие концепции, как модульное программирование, управление прерываниями, параллельная обработка и микропрограммирование. Здесь же показано, как автономные графические процессоры могут создавать сложные рисунки, освобождая основной процессор от большой вычислительной работы. Демонстрируется совместная работа графического и центрального процессоров с использованием графического устройства изображения и светового пера.

Другая, параллельная часть книги посвящена более детальному описанию основных особенностей операционных систем и других элементов системного программного обеспечения. Материал *глав 10* и *11* суммирует и унифицирует ранее рассмотренные идеи и концепции организации ЭВМ, программирование на языке Ассемблера с использованием свойств оборудования и характеристик системного программного обеспечения. Так, *в главе 10* сначала объясняется, зачем нужна операционная система, а затем описывается система общего назначения с перечислением необходимых функций и объяснением того, как они реализуются. И наконец, после того как читатель ознакомлен с внутренним строением операционной системы, приводятся сведения об

областях, в которых могут использоваться операционные системы малых компьютеров. Последняя, *11-я глава* посвящена описанию современной мультипрограммной системы. Эта система проста для понимания и построена на базе большинства идей и концепций, которые реализованы в больших операционных системах реального времени, поддерживаемых большинством разработчиков малых машин. Она представляет собой законченный продукт для применения. Поэтому пользователь путем детального анализа ее общей организации может понять, как такая базовая система может быть расширена для индивидуального объекта автоматизации.

Книга снабжена ссылками на литературу для дальнейшего изучения и многими примерами различной трудности. Часть примеров специально подготовлена для настоящего издания. Особенностью данной книги можно также считать то, что приведенные в ней листинги программ и краткие комментарии взяты из библиотеки Digital Equipment Computer User's Society (DECUS) и, следовательно, являются листингами проверенных программ.

По нашему мнению, этот вводный курс может послужить еще одной ступенькой для продвижения в науке об ЭВМ. И поэтому мы попытались обеспечить читателю необходимый (и достаточный) уровень знаний, требуемый для дальнейшего понимания вычислительных систем. В нашу цель не входило научить его писать и выполнять программы на PDP-11, хотя мы и надеемся, что он приобрел хорошие эксплуатационные навыки работы на этой машине. Скорее, мы ожидаем, что в результате усвоения материала книги читатель постигнет основные идеи ЭВМ и принципы организации вычислительных систем.

Р. Экхауз,
Л. Моррис

Глава 1

ВВЕДЕНИЕ

В течение последней четверти века ЭВМ в значительной степени изменили окружающий мир, а грядущие годы таят в себе дальнейшие, еще более значительные перемены.

Современные универсальные цифровые ЭВМ отличаются от своих прототипов завидным быстродействием, меньшими габаритами, возросшей надежностью и низкой стоимостью. Это стало возможным благодаря новым методам производства, разнообразным архитектурам и более быстрой памяти. Еще более впечатляющие результаты достигнуты в тех областях, где мы научились применять ЭВМ.

Первые большие электронные машины использовались как сверхкалькуляторы при решении сложных математических проблем, к которым раньше нельзя было подступиться вообще. В последнее время программисты находят применение ЭВМ не только для вычислений. Так, машины применяются при управлении объектами, в системах связи, для моделирования искусственного интеллекта, распознавания образов, а также для манипуляции и обработки данных. Во всех случаях вычислительная система с высокой производительностью обрабатывает огромное количество информации.

Появление мини-ЭВМ ускорило и расширило воздействие вычислительных машин на различные сферы применения. Разработка схем очень большой степени интеграции — чипов (chip) позволила создать ЭВМ на чипах, точнее, микромашинные системы, по функциям и возможностям равные мини-ЭВМ. Такие микропроцессоры удивительно экономны и особенно впечатляют в качестве интеллектуальных управляющих устройств в домашнем хозяйстве, игрушках, кинокамерах, автомобилях и других предметах широкого потребления. Это привело к тому, что роль и значение программиста вычислительной машины неизмеримо возросли.

1.1. ЧЕГО ТРЕБУЕТ ЭВМ?

Ранее говорилось, что ЭВМ может быть запрограммирована на решение любой задачи, которая поддается формализации. Решение можно разбить на серии шагов, записанных на последовательность команд машины. Однако формулировка некоторых проблем, таких, как перевод с одного естественного языка на

другой, оказывается весьма трудной. Так, например, несколько лет назад считалось, что может быть написана программа, осуществляющая перевод французского текста на английский язык. Как оказалось на самом деле, легко перевести только список отдельных французских слов, но перевести точно целое предложение трудно из-за множества оттенков значений слов и их комбинаций. По этой причине не имеет смысла пытаться общаться с ЭВМ, применяя наш обычный разговорный язык.

Для взаимодействия с ЭВМ служат специальные языки программирования с хорошо определенной структурой и синтаксисом. В частности, такие языки, как BASIC и FORTRAN, удовлетворяют предъявляемым требованиям и значительно упрощают связь с машиной. Подобные средства программирования обычно проблемно-ориентированные и содержат знакомые слова и выражения; таким образом, применяя язык программирования, можно научиться писать программы после сравнительно короткого периода тренировок. Так как большинство изготовителей ЭВМ адаптировали стандартные языки и реализовали их в составе своего программного обеспечения, программа обычно может быть выполнена на многих машинах. Программисты часто применяют FORTRAN для решения научных и инженерных проблем, в то время как BASIC — для коротких числовых расчетов. Существуют языки программирования для управления станками, автоматизации типографского набора, создания музыки, сбора данных, расширения множества машинных команд и др. Создание каждого нового языка облегчает использование вычислительных мощностей ЭВМ для решения специфической проблемы или задачи.

Кто может быть программистом? На ранней стадии программирования большинство программистов составляли математики. Однако, как будет показано дальше, в основном программирование требует только элементарного умения выполнять арифметические и логические операции. Самым главным требованием для успешного программирования можно считать умение рассуждать логически.

Быстрое расширение областей применения ЭВМ в последнее десятилетие открыло доступ к ним сотням тысяч людей и обеспечило новые благоприятные возможности для успешного использования вычислительных средств.

1.2. ОБЛАСТИ ПРИМЕНЕНИЯ ЭВМ

ЭВМ, как и любой другой механизм, применяется потому, что она решает определенные задачи лучше и эффективнее, чем человек. Вычислительная машина может оперировать бóльшим количеством информации и обработать ее быстрее. Такая скорость означает, что результаты, получаемые за недели или месяцы работы с карандашом и бумагой, могут быть получены за несколько минут машинного времени. Поэтому компьютер используется в случаях, когда выигрыш во времени может противосто-

ять его стоимости. Благодаря способности к быстрой манипуляции большими объемами данных ЭВМ, возможно, является единственным средством решения проблем в условиях ограниченного времени. Преимущества, обеспечиваемые большой скоростью и пропускной способностью, позволяют широко применять ЭВМ в мире бизнеса, индустрии и научных исследований. Основную часть областей применения ЭВМ можно отнести или к сфере бизнеса (ведение баз данных), где обычно используется способность машины хранить и быстро извлекать большие количества информации, или к научным исследованиям, требующим точности и скорости при выполнении математических вычислений. В обоих случаях используются универсальные компьютеры. Далее мы опишем несколько примеров их применения.

Проблемы проектирования. ЭВМ — очень полезный вычислительный инструмент для инженера-проектировщика. Параметры крыла для сверхзвукового самолета, например, зависят от многих факторов. Проектировщик описывает каждый из них в форме математических уравнений, записанных на каком-либо языке программирования, и ЭВМ решает эти уравнения при произвольно вводимых данных.

Научные и лабораторные эксперименты. В научных и лабораторных экспериментах вычислительные машины применяются для оценки, хранения и обработки информации, поступающей от различного типа датчиков. В частности, ЭВМ используются в телеметрических системах, где сигнал должен быть быстро записан, иначе он будет потерян. Такой вид применений требует быстрой и точной обработки как фиксированных, так и динамических ситуаций.

Управление технологическими процессами. ЭВМ является полезным инструментом для автоматического управления производством и контроля продукции. Она может быть запрограммирована для управления работой фрезерных и токарных станков и многих других устройств, где требуется более быстрая реакция, чем у человека. С помощью специальных датчиков компьютер может следить за обработкой отдельных деталей, настраивая станок в случае необходимости. Если подаваемая деталь имеет дефект, то ЭВМ отвергнет ее и начнет работу с новой.

Использование в качестве тренажера. Часто оказывается дорого, опасно и непрактично обучать большую группу людей выполнению практических функций в реальных условиях и на реальных объектах, например управлению самолетом, спутником или космическим кораблем. Машина может моделировать все требуемые условия, обеспечив многочасовую тренировку практиканта, без риска для него самого и для других, а также без использования дорогостоящего оборудования.

Применение ЭВМ для этих целей обычно требует обработки как аналоговой, так и дискретной или цифровой информации. Аналоговая информация состоит из непрерывно измеряемых значений физических величин, которые легко генерируются и

контролируются. Это, например, электрическое напряжение или вращение вала. Дискретные данные содержат дискретные числовые значения, которые представляют параметры, характеризующие объект. Обычно аналоговые величины преобразуются в эквивалентные цифровые, с тем чтобы их можно было использовать в арифметических вычислениях, выполняемых машиной. Существуют специальные гибридные ЭВМ, оперирующие одновременно как аналоговыми, так и дискретными данными.

1.3. ВОЗМОЖНОСТИ И ОГРАНИЧЕНИЯ ВЫЧИСЛИТЕЛЬНОЙ МАШИНЫ

Компьютер — это машина и, как любая другая машина, нуждается в соответствующем управлении при реализации возложенных на нее функций. Такое управление выполняется программой, и, пока эта программа не подготовлена и не размещена в памяти, ЭВМ «не знает» абсолютно ничего, даже того, как принять данные. Даже самой совершенной вычислительной машине нужно «сказать», что она должна делать. Ресурсы ЭВМ могут полностью использоваться только если известны ее возможности и ограничения:

1. *Повторение операций.* ЭВМ может исполнять одни и те же операции тысячи раз, не скучая, не уставая и не становясь небрежной.

2. *Скорость.* Машина обрабатывает информацию в чрезвычайно высоком темпе. Например, современные ЭВМ могут решать некоторые проблемы в миллион раз быстрее, чем квалифицированный математик.

3. *Гибкость.* Универсальные ЭВМ могут быть запрограммированы для решения разных проблем.

4. *Точность.* Компьютер может быть запрограммирован так, чтобы вычислять ответ с заданной степенью точности, специфицированной программистом.

5. *Интуиция.* ЭВМ не обладает интуицией. Человек способен внезапно найти решение проблемы без проработки деталей, в то время как машина может работать только выполняя заданный порядок вычислений.

1.4. АРХИТЕКТУРА, ОРГАНИЗАЦИЯ И РЕАЛИЗАЦИЯ ЭВМ

При анализе любой вычислительной системы ее полезно рассматривать на трех уровнях, наиболее известным из которых является уровень архитектуры ЭВМ.

Архитектура вычислительной машины часто определяется как множество ресурсов, доступных пользователю. Это включает, например, *систему команд, общие регистры, слово состояния процессора* и *адресное пространство.*

Что касается *организационного уровня,* то он находится «ниже» архитектурного и определяет организацию связи процес-

сора, памяти, устройств ввода-вывода в одной функционирующей системе. В частности, он требуется при рассмотрении микромашин, которые содержат множество элементов, определяющих архитектуру. На этом уровне можно получить ответ на вопрос: является ли микромашина микропрограммируемой? Анализ связей на организационном уровне позволяет понять, как различные ЭВМ с одинаковой архитектурой могут быть организованы таким образом, что дают возможность пользователю осуществлять выбор скорости обработки и/или размера памяти задачи.

Наконец, на *уровне реализации* ЭВМ воспринимается как система проводников, кабелей, полупроводниковых чипов и дискретных компонентов. Если же нас интересуют выбор технологии полупроводников, временные фазы и прохождение сигналов между различными элементами, а также другие электрические параметры, то мы можем считать себя «дизайнерами ЭВМ».

Гл. 2 посвящена краткому описанию архитектуры и общей организации цифровой машины, ее действиям по манипуляции данными, а также природе машинных программ. Глава включает также введение в базовую структуру оборудования и программного обеспечения машин и описание основных арифметических и логических операций. Обсуждаются альтернативные подходы к адресации операндов, различные способы спецификации адреса и манипуляции адресами. Эта информация дает основы для понимания и применения ЭВМ, а также является предпосылкой для программирования на уровне машинного языка и языка Ассемблера. В педагогических целях представляемый материал не привязывается к специфике PDP-11. Вместо этого обсуждается машина, *подобная* PDP-11, но без конкретностей, присущих реальной машине.

В гл. 3 подробно анализируется специфика архитектуры PDP-11 и *организация* различных машин этого семейства. Рассматриваются форматы команд и данных, специфичные для таких машин, с тем чтобы пользователь мог писать реальные программы для реальных ЭВМ. Ниже, в гл. 6, будет обстоятельно описана одна из PDP-11 (11/60), включая ее организацию и микропрограммирование.

Хотя данная книга иллюстрирует функционирование только одного представителя семейства мини/микроЭВМ — PDP-11, однако большую часть материала можно применять гораздо шире. Некоторые понятия, такие, как базовые арифметические команды (например, ADD, SUBTRACT и т. д.), похожи по назначению для всех компьютеров. С другой стороны, режимы адресации часто являются специфичными для конкретной ЭВМ. Однако в обоих случаях материал книги изложен так, что знакомство со спецификой одной архитектуры позволяет быстро перейти к другой.

И наконец, признано, что архитектура PDP-11 является *мощной* и *бит-эффективной* (bit-efficient) для вычислительных операций благодаря гибкости формирования адресов с помощью

регистров. Это признание подтверждается сравнительным изучением архитектуры машин и их широким распространением и признанием у пользователей. Статистика показывает, что пользователь желает иметь или PDP-11, или машину, подобную ей (например, M6800, MC68000, Z8000). Итак, если надо изучать реальную машину, то выбор одной из моделей семейства PDP-11 вполне обоснован.

Типичная мини-компьютерная система PDP-11 вместе с отдельными периферийными устройствами может быть рассмотрена на примере ЭВМ PDP-11/60. Основная масса данных в такой системе хранится на гибких дисках малой емкости или на дисковых пакетах средних габаритов. В качестве главной системной консоли в ней использован телепринтер со скоростью передачи 30 символов/с. Графический процессор обеспечивает отображение как текста, так и графического образа. Подключенные аналоговые и цифровые интерфейсы, позволяющие вести обмен информацией как последовательно, так и параллельно, обеспечивают ведение диалога и обработку данных в режиме реального времени.

Глава 2

ТЕХНИЧЕСКИЕ СРЕДСТВА И ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ЭВМ

2.1. ОСНОВНЫЕ ЭЛЕМЕНТЫ ТЕХНИЧЕСКИХ СРЕДСТВ

Обобщенная структура, присущая большинству современных вычислительных машин, показана на рис. 2.1.

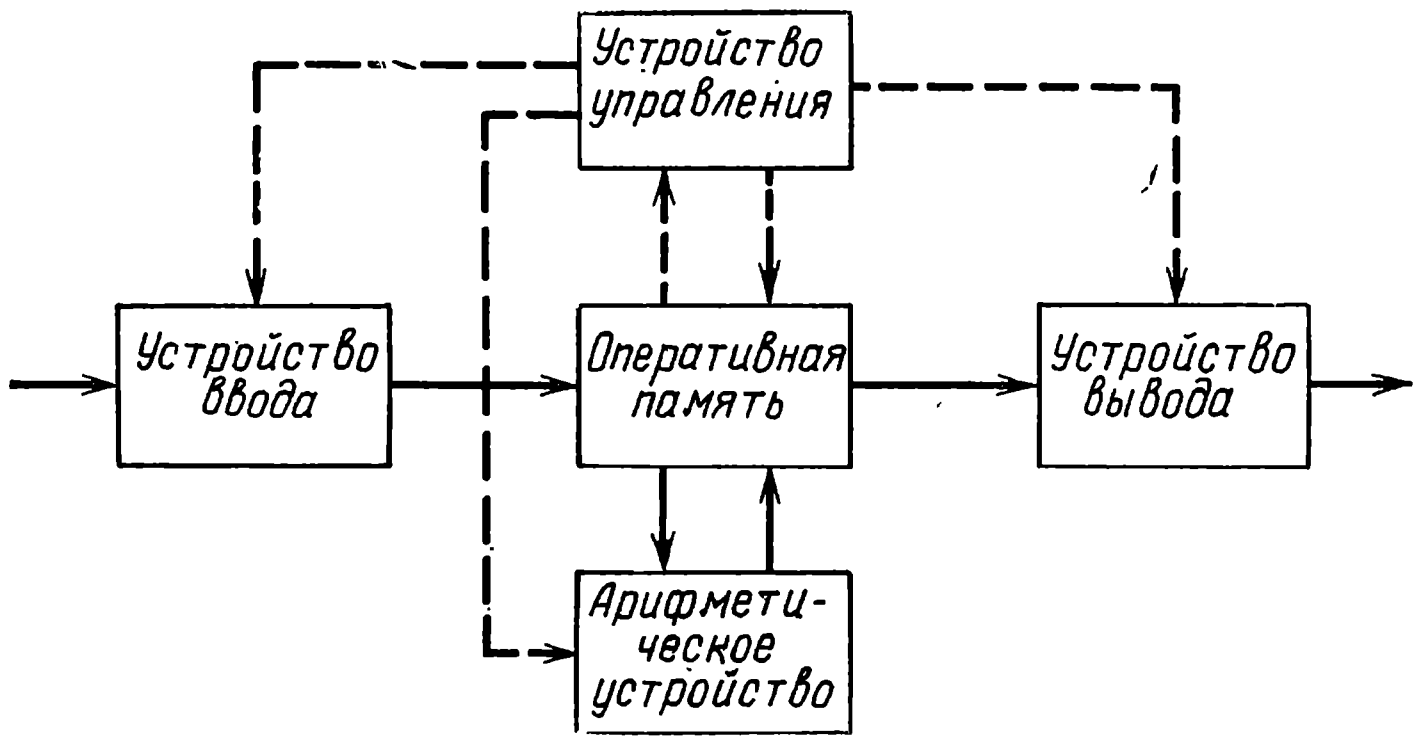


Рис. 2.1. Общая организация ЭВМ

Среди функций, реализуемых в машине, главное место занимает выполнение определенных типов логических и арифметических операций. Для этой цели используется блок, называемый *арифметическим* и *логическим устройством* (ALU). Второй блок — *устройство управления* — обеспечивает функционирование ALU, сообщая ему нужную последовательность управляющих воздействий.

ALU может выполнять математические операции, следовательно, при решении задачи может возникнуть необходимость в сохранении промежуточного результата на то время, пока это устройство используется для решения других частей задачи. Запомненный результат может быть неоднократно использован в дальнейшем. В процессе работы машины ALU, устройству управления и ряду других устройств необходим непосредственный доступ к информации, с которой они работают. Удовлетворению

этих требований служит еще одно устройство ЭВМ — *оперативное запоминающее устройство,* или просто *память.*

Для того чтобы человек мог использовать машину в соответствии со своими потребностями, в составе ЭВМ должны быть устройства, обеспечивающие его взаимодействие с системой; такие устройства называются *устройствами ввода-вывода* (I/O).

2.1.1. Арифметическое и логическое устройство

Собственно вычисления в цифровой вычислительной машине осуществляет ALU. Выполнение этой работы базируется на использовании его логических схем. В первых моделях эти схемы реализовывались на реле и переключателях. Современные вычислительные машины строятся на основе интегральных микросхем, каждая из которых состоит из тысяч транзисторов. Микросхемы имеют малые габариты, высокое быстродействие и хорошие экономические показатели. На сегодняшнем технологическом уровне целая ЭВМ может быть размещена на одном чипе из полупроводникового материала.

2.1.2. Устройство управления

Устройство управления — управляющий центр ЭВМ. Это устройство оценивает поток входной информации и принимает решение о том, когда и как выполнять обработку. Оно задает арифметическому устройству последовательность работы и источник получения необходимой информации и, зная момент завершения вычислений, сообщает в ALU, как поступить с результатами, а также что делать дальше.

Управляющее воздействие для ALU устройство управления вырабатывает на основании интерпретации последовательности *команд.* Команда содержит информацию о том, что нужно делать (*код операции*) и где расположены обрабатываемые данные (*адреса операндов*). Последовательность команд для устройства управления, хранящаяся в памяти ЭВМ, называется *программой.*

Выполнив очередную выбранную из памяти команду, устройство управления обращается за следующей, для указания адреса которой используются два способа. Во-первых, в текущей команде можно указывать адрес следующей. Но более распространен второй способ, основанный на использовании специального *счетчика команд* (PC), который содержит адрес следующей исполняемой команды.

Команды выполняются во времени последовательно, в порядке, определенном содержащейся в них информацией, либо в соответствии со значением счетчика команд. В наиболее совершенных машинах допускается возможность совмещения исполнения данной команды с выборкой из памяти следующей, но реализация этой функции происходит вне сферы деятельности программиста.

2.1.3. Память

Устройство памяти, называемое иногда основной или *оперативной памятью*, хранит информацию, предназначенную для устройства управления (команды) и для арифметического устройства (данные). Размещение в одной памяти и команд, и данных представляет собой фундаментальную идею, лежащую в основе ЭВМ с хранимой программой. Содержимое любой ячейки памяти, указанной счетчиком команд, может быть интерпретировано как команда, если оно имеет соответствующее значение.

Оперативная память ЭВМ строится на основе ферритовых сердечников или полупроводниковых микросхем, ее можно назвать *первичной*. Внешние запоминающие устройства, такие, как магнитные диски или ленты,— это *вторичная память*. Размеры и время доступа к памяти у различных моделей ЭВМ могут сильно различаться.

Память ЭВМ состоит из отдельных ячеек, каждая из которых может содержать элемент данных, часть команды или всю ее.

Памятью с прямым доступом называют систему, в которой время доступа ко всем ячейкам одинаково. Память, в которой время доступа определяется взаимным расположением в пространстве носителя информации и считывающего устройства, называют *памятью с последовательным доступом*. Например, магнитная лента должна быть перемотана до обнаружения нужной информации, и только тогда эта информация может быть прочитана.

2.1.4. Устройства ввода и вывода

Устройства ввода служат для обеспечения ЭВМ как обрабатываемыми данными, так и командами, указывающими способ обработки. Спектр характеристик таких устройств очень широк. На микроЭВМ для ввода достаточно иметь только клавиатуру пульта. Быстрее данные могут быть введены с перфокарт. Для мини-ЭВМ наиболее типично применение трех типов устройств ввода: перфолент, магнитных лент и магнитных дисков; кроме этих основных устройств могут использоваться дополнительные. В системах реального времени это аналого-цифровые преобразователи, в многомашинных комплексах — средства для последовательной или параллельной передачи информации из одной машины в другую. Интерактивные системы оснащаются дисплейным процессором с экраном, световым пером, цифровым планшетом, координатной ручкой управления курсором и т. д.

Устройства вывода записывают результаты работы ЭВМ. Запись может быть выполнена в виде «твердой копии» на бумаге при помощи печатающего устройства или в виде «мягкой копии» на экране видеотерминала. При использовании ЭВМ в системе управления технологическим объектом выходные сигналы передаются на устройство регулирования параметров. Для получения твердых копий используются такие носители, как перфоленты, перфокарты, магнитные ленты и диски.

2.1.5. Организация памяти

Будем называть *обрабатываемыми информационными единицами* элементы хранимой в машине информации, вне зависимости от того, команды это или данные. Каждая информационная единица содержится в одной ячейке, имеющей индивидуальный адрес. Использование адреса обеспечивает однозначный доступ к информационной единице. Итак, ячейка памяти имеет две характеристики:

1) *адрес* — числовое значение, ассоциированное с расположением ячейки в памяти;

2) *содержимое* — число, физически хранимое в данной ячейке. Для понимания работы машины очень важно не путать адрес (имя) с содержимым (информацией).

Информационная единица может быть одной двоичной цифрой — *битом*, набором бит — *байтом* или набором байт — *словом*. В первых моделях ЭВМ из-за экономических ограничений система команд была развита слабо, поэтому число типов информационных единиц, доступных из команды, тоже было ограничено. Это приводило к тому, что при проектировании машины заранее фиксировался размер информационной единицы, к которой можно адресоваться из команды, в соответствии со сферой применения машины. Например, для обработки экономической информации, включающей как символьные, так и цифровые данные, использовались байтовые машины. В то же время для выполнения научных расчетов, где требуется высокая точность, использовались машины, оперирующие целыми словами.

Принципиальное различие между битовыми, байтовыми и словными машинами состоит в размере минимально адресуемой информационной единицы. Для битовой машины это одна двоичная цифра, т. е. информационная единица принимает значение либо 0, либо 1. Для байтовой машины это один из знаков набора символов данной машины, буква, цифра или специальный знак. Наконец, для словной машины это числовое значение, пределы которого определяются характеристиками ЭВМ.

Конечно, можно внутри слова ввести условное деление на символы, но при этом словная машина все равно не способна адресовать отдельный символ, а допускает адресацию только всего слова целиком. Однако такое деление позволяет представить слово как набор символов, где каждому знаку соответствует однозначный код.

По приведенной выше классификации PDP-11 — байтовая машина, поскольку для доступа к информации в ней используются адреса байт. Тем не менее она спроектирована так, что позволяет удобно оперировать и 16-битовыми словами (2 последовательных байта), и 32-битовыми двойными словами (2 последовательных слова), и 64-битовыми словами учетверенной длины (2 последовательных слова двойной длины). Достижения современной микроэлектроники сделали вполне обычным наличие в ЭВМ такого

развитого набора команд, который позволяет работать с большим числом различных информационных единиц

2.2. ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ЭВМ. ВВЕДЕНИЕ В ПРОГРАММИРОВАНИЕ

Современные ЭВМ могут воспринимать и запоминать информацию, выполнять вычисления, принимать решения на основе этих вычислений, наконец, получать окончательное решение задачи и передавать результаты решения во внешний мир. Однако всего этого машина не может делать без указаний человека, каждый ее шаг должен быть заранее подготовлен пользователем. Для этого нужно написать последовательность команд, в соответствии с которой будет работать машина, т. е. составить *программу*. Создание программы основывается на *алгоритме* решения задачи, в соответствии с ним создается последовательность команд, которая в дальнейшем помещается в память и выполняется.

2.2.1. Этапы программирования

Для успешного решения задачи на ЭВМ программист должен пройти следующие 6 этапов программирования:

1. Постановка задачи.
2. Выбор наиболее приемлемого алгоритма.
3. Определение типов входных и выходных данных.
4. Проектирование и анализ решения, в том числе составление блок-схем, описаний и т. д.
5. Кодирование алгоритма на языке программирования.
6. Проверка программы.

В большинстве случаев постановка задачи не всегда очевидна, однако, если она неверна, можно потерять много сил и средств. Постановка элементарной задачи суммирования четырех чисел тривиальна, но при решении, например, такой задачи, как создание системы тестирования полупроводниковых изделий, необходимо уделить большое внимание ее постановке. Итак, на первом этапе необходимо ответить на вопрос: какой точно должна быть программа?

На втором этапе делается выбор из множества возможных методов решения задачи, причем часто он может определяться конкретными свойствами используемой вычислительной системы. Выбрав метод решения и зная возможности данной вычислительной системы, программист должен разработать вычислительный алгоритм.

Между выбором алгоритма и проектированием решения программист должен определить и описать используемую в программе *информацию,* ее *типы* и *количество* Данные могут иметь сложную и подчас неявную структуру, поэтому необходим их достаточно детальный анализ.

Проектирование и *анализ* решения состоят в определении отдельных шагов решения и организации их в логическую последовательность для исполнения. Эту последовательность графически можно представить в форме блок-схем. Блок-схемы являются эффективным средством визуализации процесса решения и дают возможность дальнейшего анализа и оценки возможных альтернативных подходов.

После выполнения этапа 4 осуществляется *кодирование программы на определенном языке программирования.* Зачастую именно этот этап ошибочно называют программированием, хотя, по существу, кодирование — это только часть всего процесса создания программы. Возможно, что закодированная и введенная в память машины программа может использоваться для решения задачи, однако, такое случается не часто. Весьма небольшая часть программ при первом вводе в ЭВМ работает так, как предполагалось, и поэтому, как правило, программист вынужден перейти к шестому этапу — этапу *проверки и отладки программы.*

Для исправления ошибок необходимо тщательно проанализировать текст программы. Дело в том, что машина очень точно исполняет то, что от нее затребовано, но она не способна выполнить ничего сверх того. Программист не может себе позволить сказать, как это мы делаем в обыденной жизни: «Сделай то, что я подразумеваю». Если программа содержит ошибочно записанные команды или в ней пропущены какие-либо команды, то полученные результаты бывают весьма неожиданными. Существует множество подходов к организации тестирования и отладки программ, но вне зависимости от того, какой из них выбран, важно, чтобы он осуществлялся последовательно и пунктуально. Некоторые широко применяемые приемы отладки будут описаны ниже.

2.2.2. Документирование программ

Ниже в качестве примера приводится программа сложения трех чисел. Для решения она требует написания всего нескольких команд, однако нужный результат быстрее может быть получен с помощью карманного калькулятора или на листе бумаги при помощи карандаша. Безусловно, на ЭВМ целесообразно решать более сложные задачи, выполнение которых требует много времени или вообще невозможно без применения машинных методов. Именно сложностью решаемых на ЭВМ задач определяется необходимость их документирования.

Блок-схемы. Программирование в общем случае представляет собой длительный и трудоемкий процесс, причем одна из трудностей состоит в том, что кодирование программы на языке программирования на основании описания задачи на естественном языке или в виде математического выражения довольно неудобно, поскольку по таким описаниям невозможно проследить последовательность решения. Для упрощения написания программ ис-

пользуется такое средство, как *блок-схемы.* Блок-схемы — это графическое средство представления поставленной задачи, с помощью которого можно продемонстрировать последовательность действий, выполняемых при ее решении. Блок-схемы могут оказаться полезными и при отладке программ.

При построении блок-схем используется набор определенных графических символов, соединяемых между собой линиями. Символами обозначаются выполняемые операции, линиями задается их последовательность. Форма любого графического символа соответствует определенной функции. На рис. 2.2 представлены некоторые символы, применяемые при разработке блок-схемы; они соответствуют стандартам ANSI/ISO.

Точки входа и выхода	Символ в форме овала служит для обозначения начала, конца или точки прерывания программы. Нужная запись включается внутрь символа.
Обработка	Прямоугольным символом обозначается функция обработки. Обработка — это одна или несколько команд, служащих для решения некоторой задачи. Внутрь символа вписывается краткое содержание обработки.
Решение	Ромб используется для обозначения таких мест в программе, где принимается решение о последовательности дальнейших действий. Внутрь ромба вписываются анализируемые условия, а возможные результаты анализа обозначаются на выходящих из ромба линиях.
Коннектор	Небольшой кружок с однозначной меткой приписывается к концу отрезка линии и обозначает соединение с другой линией, имеющей кружок с такой же меткой.

Рис. 2.2. Символы в блок-схемах

Ниже приведены два примера использования блок-схем. В первом из них выполняется сложение трех чисел, а в примере 2 три числа упорядочиваются по возрастанию.

Пример 1. Программа без переходов. Здесь иллюстрируется линейная программа, не содержащая переходов. На рис. 2.3 показана выполняемая последовательность шагов, также линейная. По окончании работы в ячейке D будет находиться значение выражения $A + B + C$.

Рис. 2.3. Сложение трех чисел

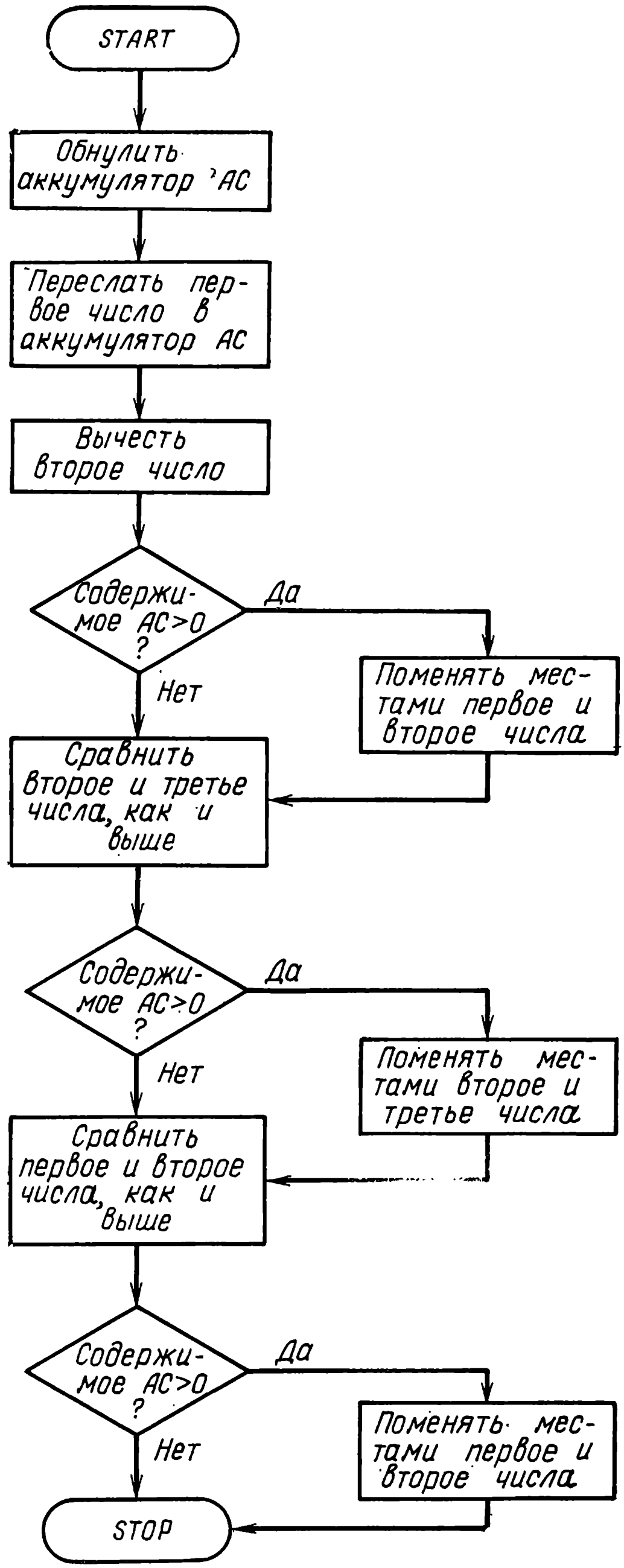


Рис. 2.4. Упорядочивание трех чисел в возрастающем порядке

Пример 2. Программа с переходами. В этом примере требуется упорядочить по возрастанию три числа (рис. 2.4). Для изменения порядка следования чисел программа должна содержать переходы. Более подробно переходы как средство программирования рассматриваются позже. В данном примере арифметическая операция вычитания выполняется в аккумуляторе (*AC*), которым может быть ячейка памяти или регистр, связанный с ALU. Первоначально *AC* должен быть обнулен.

Дополнительные методы документирования. Хорошие программы используются достаточно долго, и со временем они могут модифицироваться и исправляться уже не тем программистом, который их создавал. Поэтому необходима документация, позволяющая работать с ними любому программисту. Развернутая и точная блок-схема является хорошей формой документирования, но

	Характеристики	Правило		
Условия	Число ног	4	4	2
	Длина шеи	> 4′	< 4′	< 4′
	Маленький нос	Да	Нет	Да
Действия	Животное — жираф, давайте ему	Сено		
	Животное — слон, давайте ему		Арахис	
	Животное — петух, давайте ему			Зерно

Рис. 2.5. Таблица решений

на ней не всегда удается должным образом отразить намерения автора. По назначению блок-схемам аналогичны описания входных данных, ограничений, накладываемых использованными алгоритмами, и взаимосвязей между отдельными модулями программы.

Во многих случаях лучшим средством для описания служит естественный язык. В исходный текст программы следует включать заголовки и комментарии, причем последние стоит писать к каждой команде, чтобы неподготовленный пользователь мог разобраться с работой программы, не анализируя ее текст, а пользуясь для этого только комментариями.

Таблицы решений — еще одно достаточно популярное средство документирования программ. Такая таблица в графической форме отражает отношения между набором условий и набором операций, соответствующих определенному набору правил. Это

достаточно компактное средство, по объему таблицы обычно не превышают тысячи слов и не зависят от типа ЭВМ и языка программирования.

На рис. 2.5 приведен пример таблицы решений для классификации животных по числу ног, длине шеи и размеру носа для выбора соответствующего питания. Хотя для иллюстрации использован тривиальный пример, однако следует отметить, что этот метод наиболее полезен при описании сложных процессов. Одно из основных достоинств таблиц решений состоит в том, что они легко конструируются, модифицируются и легки для понимания.

Этот метод не лишен недостатков, и здесь в первую очередь стоит отметить следующее. Во-первых, программирование таблиц решений представляет собой довольно сложную задачу, несмотря на существование специально разработанных языков. Во-вторых, в тех случаях, когда ситуация не слишком сложна, такой аппарат может оказаться слишком мощным и, следовательно, обременительным. При этом вполне достаточно используемых на блок-схемах обозначений переходов. Что касается данного пособия, ориентированного на изучение ЭВМ на уровне языка Ассемблера, то на этом уровне недостатки таблиц решений не позволяют их использовать.

2.2.3. Кодирование программ

Машине понятен единственный язык — язык двоичных цифр. Хранимые в памяти команды и данные представляются в двоичной форме, и только в таком виде они могут быть интерпретированы ЭВМ при выполнении программ, инструкции на естественном языке машина, к сожалению, понимать не в состоянии.

Каждая машина имеет некоторый свой собственный набор двоичных кодов, который воспринимается ею как набор команд, т. е. машина устроена так, что на каждое число из этого набора она реагирует определенным образом. Поскольку внутреннее представление и команд, и данных одинаково, ЭВМ может воспринять одну и ту же комбинацию нулей и единиц по-разному, в соответствии с тем, куда эта комбинация включена программистом в программу. Собственно двоичные числа в практике программирования используются редко, обычно их заменяют более удобными восьмеричными или шестнадцатеричными.

Кодирование программы в форме последовательности восьмеричных чисел несколько облегчает программирование, однако оно все еще очень трудоемко. Программист вынужден запоминать целый набор восьмеричных цифр, представляющих коды операций, причем между цифрами и операциями не существует никакой логической связи. Хотя писать программу в восьмеричных кодах трудно, еще труднее ее отлаживать, потому что отсутствуют методы, облегчающие заполнение соответствия восьмеричных цифр функциям, которые они обозначают.

Программирование может быть существенно облегчено, если сопоставить командам их символьное мнемоническое представление и обозначить каждую из них 2—5 символами. Мнемоника дает возможность в сокращенной форме выражать выполняемую командой функцию.

2.2.4. Концепция хранимой программы

Концепцией хранимой программы называют идею использования памяти для хранения и команд, и данных. Хотя сейчас это кажется вполне естественным, в первых экспериментальных машинах такое совмещение не допускалось. Впервые это было высказано и затем внедрено в вычислительной системе Джоном фон Нейманом в 1940-г.

Заслуга фон Неймана в том, что его идея позволяла интерпретировать команды как данные, и при этом появилась возможность выполнять арифметические операции над командами. Еще более важно то, что она дала возможность быстро изменять порядок выполнения команд в программе с помощью команд условного или безусловного перехода. Таким образом, работа фон Неймана является основной теоретической предпосылкой для создания вычислительных машин, и его имя теперь связано с концепцией *хранимой программы*, или *концепцией фон Неймана.*

2.2.5. Типы команд

Существует четыре основных типа команд: арифметические и логические команды, команды управления, команды внутренней пересылки данных и команды ввода-вывода.

Используя *арифметические* и *логические* команды, ЭВМ выполняют четыре действия арифметики и ряд логических операций. Некоторые машины, например PDP-11/05, непосредственно могут проделать только сложение и вычитание, а умножение и деление осуществляется многократным исполнением сложения и вычитания, другие, например IBM 1620, не могут выполнять даже и сложение, они используют для получения результатов специальные таблицы.

Команды второго типа — *команды управления* — служат для определения порядка, в котором выполняются команды других типов. Эти команды позволяют переходить из одной точки программы в другую, обходить некоторую последовательность команд по результатам выполнения, допустим, арифметического действия, и ряд других аналогичных операций.

Третья группа команд — *команды внутренней пересылки данных* — дает возможность перемещать целиком слова или байты при требуемых в большинстве программ внутренних пересылках данных. Кроме того, команды этого типа позволяют выполнять сдвиговые операции над битами одного слова.

Последний класс команд — *команды ввода* и *вывода.* Команды ввода указывают устройству управления на необходимость передать информацию от некоторого внешнего по отношению к системе устройства в память для хранения внутри системы. Команды вывода выполняют передачу информации в обратном направлении. Кроме того, команды ввода-вывода можно использовать для получения информации о состоянии устройства и для передачи ему управляющих воздействий.

2.3. АРХИТЕКТУРА АДРЕСАЦИИ ОПЕРАНДОВ

Два из основных устройств, обозначенных на рис. 2.1,— ALU и устройство управления — вместе образуют *центральный процессор* (CPU). Функция CPU состоит в выборе из памяти очередной команды, ее декодировании, выборе из памяти необходимых для исполнения команды операндов и, наконец, в выполнении команды. Таким образом, для работы CPU команды, поступающие в него, должны содержать следующую информацию:

1. Код выполняемой операции — число, которое имеет мнемоническое обозначение, например ADD — сложить, SUB — вычесть, MUL — умножить и т. д.
2. Адреса операндов и результата.
3. Адрес следующей исполняемой команды.

Поскольку при выполнении большинства арифметических операций используются два операнда и еще требуется адрес, куда поместить результат и адрес следующей команды, формат команды должен включать код операции и четыре адреса. В этом разделе рассматриваются различные типы архитектур машин, т. е. возможные форматы команд ЭВМ, которые удовлетворяют поставленным выше требованиям. Далее, в разд. 2.4, будет показано, как используются эти четыре адреса. Заметим, что символы «*», «/», «**» в дальнейшем употребляются для обозначения соответственно операций умножения, деления и возведения в степень.

2.3.1. Машины с адресацией три-плюс-один и трехадресные машины

На рис. 2.6 представлен формат команды машины с адресацией типа три-плюс-один (3 + 1).

Код операции	Адрес операнда 1	Адрес операнда 2	Адрес результата 3	Адрес следующей команды

Рис. 2.6. Формат команды с адресацией три-плюс-один

Символически такую команду можно представить так:

```
ADD      X,Y,Z,W
```

символический код операции — *символические имена ячеек памяти*

Это означает: сложить содержимое ячейки X с содержимым ячейки Y, результат поместить в ячейку Z и выбрать следующую команду из ячейки W. Если условиться, что «(X)» обозначает «содержимое ячейки памяти», то сказанное можно переписать так:

```
(Z) ← (X) + (Y)  & (W)
```

Здесь символ «←» означает «присвоить» или «сделать равным», а «&» — «взять следующую команду из ячейки». Форма, в которой представлена команда сложения, называется *инфиксной нотацией*; при этом выполняемая операция заключается между используемыми в ней операндами.

Рассмотрим теперь последовательность команд, необходимую для реализации следующего выражения на Фортране:

```
A = (B*C) - (D*E)
```

Команды, используемые для решения этой задачи, размещаются в ячейках памяти от 11 до 14. Используя символический формат записи и инфиксную нотацию, получим:

Символический формат	*Инфиксная нотация*
I1: MUL B,C,T1,I2	I1: (T1) ← (B)*(C) & (I2)
I2: MUL D,E,T2,I3	I2: (T2) ← (D)*(E) & (I3)
I3: SUB T2,T1,A,I4	I3: (A) ← (T1)-(T2) & (I4)
I4: ...	I4: ...

Ячейки T1 и T2 используются для временного хранения промежуточных результатов. В последующих разделах мы будем использовать символический формат и инфиксную нотацию при рассмотрении работы команд.

Анализируя внимательно метод адресации 3 + 1, можно обнаружить два интересных момента. Во-первых, при его использовании нет необходимости размещать последовательно в памяти те команды, которые в программе должны выполняться последовательно, хотя в примере они следуют одна за другой. Во-вторых, поскольку адреса всех элементов данных, участвующих в любой арифметической операции, заданы явно, нет необходимости в каких-либо технических средствах для хранения промежуточных результатов. При использовании трехадресной машины любая арифметическая операция завершается при выполнении одной команды.

Следует пояснить, почему требование располагать выполняемые команды в памяти последовательно не является обязательным.

В основном программы пишутся в последовательном порядке, поэтому вполне естественно предположить, что и машина могла бы выполнять команды тоже последовательно. Для дальнейших выкладок сделаем небольшой экскурс в историю развития вычислительной техники. До того как наступила эпоха машин с оперативной памятью на магнитных сердечниках или интегральных микросхемах, где время доступа не зависит от адреса ячейки, использовались системы памяти на ртутных линиях задержки или магнитные барабаны, которые не имели произвольного доступа. В системах памяти этого типа время доступа к ячейке, следующей после данной, было больше, чем к любой другой, поэтому программа с последовательным выполнением команд проигрывала в быстродействии программам, в которых время исполнения оптимизировалось указанием в качестве следующей той ячейки памяти, которая в данный момент была доступна быстрее. (Классическим примером машины такого типа является IBM 650, правда, у нее адресация 1 + 1.)

Однако указывать адрес следующей выполняемой команды в большинстве машин не требуется, поскольку повсеместно используется память с произвольным доступом. В них имеется указатель, который называется «счетчиком команд» (PC). Счетчик команд исполняет функцию стрелки, всегда указывающей на подлежащую выполнению команду. Аппаратно PC — это регистр, хранящий адрес текущей команды; он модифицируется в процессе ее выполнения: в нем формируется адрес следующей.

Стоимость памяти составляет существенную часть стоимости всей машины, она во многом определяется длиной машинного слова, поэтому если в формате машинной команды один адрес заменить регистром, то можно получить просто трехадресную машину и добиться определенной экономии. Стремясь к дальнейшей экономии, удалим из машинной команды еще один операнд и получим более эффективную по стоимости двухадресную машину.

2.3.2. Двухадресные машины

На первый взгляд потеря одного адреса кажется более значительной, чем это есть на самом деле. Зачастую нет необходимости в резервировании отдельной ячейки, чтобы туда поместить результат, поскольку можно использовать адрес второго операнда для хранения результата. Допустим, что нужно решить задачу сложения содержимого ячеек A и B, а результат поместить в ячейку C. Для трехадресной машины пришлось бы выполнить команду следующего вида:

```
ADD     A,B,C                    (C) ← (A) + (B)
```

Совмещение адреса результата с адресом второго операнда позволяет сэкономить по одному адресу в каждой команде. Та же за-

дача при решении ее на двухадресной машине выглядела бы так:

```
MOV     A,C          (C) ← (A)
ADD     B,C          (C) ← (C) + (B)
```

Команда «MOV» перемещает операнд из одной ячейки (*источника*) в другую (*приемник*). Заметьте, что при этом старое содержимое ячейки C уничтожается. Точно так же выражение A = (B * C) — (D * E) может быть переписано:

```
MUL     B,C          (C) ← (B)*(C)
MUL     D,E          (E) ← (D)*(E)
SUB     E,C          (C) ← (C)-(E)
MOV     C,A          (A) ← (C)
```

В приведенном выше примере значения результатов арифметических операций накладываются на значения операндов C и E, значения которых будут потеряны. Это является одним из следствий использования двухадресной системы команд.

Команды с двумя адресами используются и в больших, и в малых машинах. Данной архитектурной особенностью обладают такие машины, как PDP-11, VAX-11, MC 68000, Z 8000, IBM 1620, UNIVAC 1105, IBM 370, и ряд других.

2.3.3. Одноадресные машины

С целью дальнейшего уменьшения стоимости ЭВМ из команды можно удалить еще один адрес. Тогда единственный оставшийся должен указывать и на источник, и на приемник, но поскольку в большинстве арифметических операций применяются два операнда, то необходимо предусмотреть в *одноадресной* машине некоторое аппаратное средство для задания второго операнда. Обычно эту функцию выполняет *аккумулятор.*

Использование аккумулятора состоит в том, что при выполнении арифметических операций с его содержимым производится сложение или из его содержимого — вычитание и т. д. Следовательно, в состав системы команд одноадресной машины необходимо включить команды инициализации начального значения аккумулятора и сохранения его значения. Этой цели служат команды *загрузить в аккумулятор* (LAC) и *освободить аккумулятор* (DAC).

Ниже приведен пример программы для вычисления на одноадресной машине уже известного выражения:

A = (B*C) - (D*E)

```
LAC     D            (ACC) ← (D)
MUL     E            (ACC) ← (ACC)*(E)
DAC     T1           (T1)  ← (ACC)
LAC     B            (ACC) ← (B)
MUL     C            (ACC) ← (ACC)*(C)
SUB     T1           (ACC) ← (ACC)-(T1)
DAC     A            (A)   ← (ACC)
```

где T1 обозначает временно используемую ячейку, ACC — аккумулятор. Можно было заметить, что если бы второе умножение было выполнено в начале, то дополнительно потребовались бы одна временная ячейка и две команды — DAC и LAC.

По одноадресной архитектурной схеме построены машины PDP-8, PDP-15, IBM 1130, IBM 7090, CDC 3600 и др.

2.3.4. Машины с общими регистрами

Приняв идею ЭВМ с одним аккумулятором, ее легко можно развить в концепцию машины с несколькими такими элементами. Применение нескольких аккумуляторов для хранения промежуточных результатов позволяет уменьшить потребность во временно используемых ячейках. Кроме того, *прямое* обращение к аккумуляторам из ALU позволяет существенно повысить быстродействие.

Допустим, что мы работаем на машине с двумя аккумуляторами, система команд которой включает такие команды, как «загрузить сумматор 1» (LA1), «освободить сумматор 2» (DA2), «умножить содержимое аккумулятора 1 на содержимое указанной ячейки памяти и оставить результат в аккумуляторе 1» и т. д.; тогда представление фортрановского выражения A = (B * C) — (D * E) можно записать так:

```
LA1     D          (ACC1) ← (D)
MP1     E          (ACC1) ← (ACC1)*(E)
LA2     B          (ACC2) ← (B)
MP2     C          (ACC2) ← (ACC2)*(C)
SB2     ACC1       (ACC2) ← (ACC2)-(ACC1)
DA2     A          (A)    ← (ACC2)
```

ЭВМ, имеющие в своем составе несколько аккумуляторов, называют машинами с *регистрами общего назначения*; регистры в них используются не только в качестве аккумуляторов, но и для *индексации* элементов массивов, вычисления числа повторений *циклов*, для *указания* некоторого места в памяти и т. д.

Хотя первоначально идея применения регистров общего назначения была выдвинута в приложении к одноадресным машинам, в дальнейшем она развивалась и в машинах с другими методами адресации. Таким образом, машины с регистрами общего назначения могут быть и двух-, и трехадресными, они используют регистры вместо ячеек памяти для хранения операндов. Например, в машине PDP-11, которая является двухадресной, имеются такие команды сложения, как «сложение память — регистр», «сложение регистр — память», «сложение регистр — регистр» и «сложение память — память». Они могут быть записаны следующим образом:

```
«Сложение память — регистр»     ADD A,Reg      (Reg)  ← (Reg)+(A)
«Сложение регистр — память»     ADD Reg,A      (A)    ← (A)+(Reg)
«Сложение регистр — регистр»    ADD Reg1,Reg2  (Reg2) ← (Reg2)+(Reg1)
«Сложение память — память»      ADD A,B        (B)    ← (B)+(A)
```

где «A» и «B» обозначают ячейки памяти, а «Reg» служит для обозначения одного из регистров общего назначения.

2.3.5. Безадресная машина

Можно ли представить себе машину вообще без адресов? Да, но «да» с определенными ограничениями. Такой машине нужен некоторый механизм, который бы неявным образом снабжал операндами арифметические операции. Механизм этот называется *стеком.*

Стек можно представить себе как упорядоченную последовательность ячеек памяти или регистров, в которой расположен первый элемент, затем следует второй и т. д. Однако только первый из них доступен при выполнении арифметических операций.

Чтение операндов из стека обозначим как процедуру *выборки.* При ее выполнении выбирается первый элемент, записанный в вершине стека, и все остальные сдвигаются на одну позицию

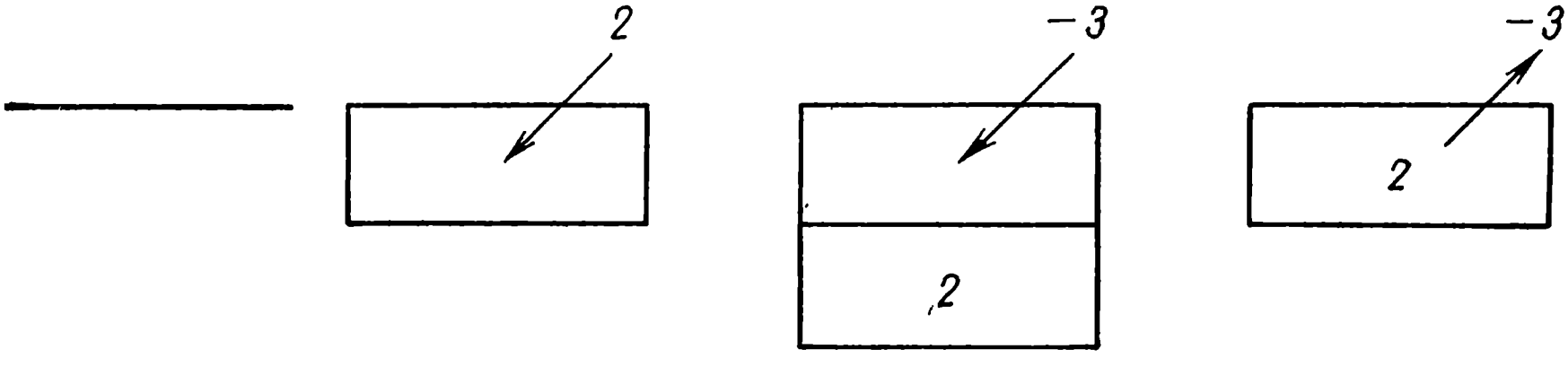


Рис. 2.7. Выполнение операций занесения и выборки из стека

вверх, т. е. второй элемент становится первым, третий вторым... Обратную операцию записи назовем *занесением.* Когда она выполняется, все элементы стека смещаются на одну позицию вниз, и на освободившуюся вершину помещается новый элемент. На рис. 2.7 приведены примеры записи и чтения из стека, который первоначально был пуст.

Арифметическая операция сложения в таком случае может быть описана так: «Занести в стек первый операнд, занести в стек второй операнд, выбрать оба операнда в сумматор, занести в стек результат». При первичном рассмотрении может показаться, что исполнение стековых операций требует постоянного перемещения хранящихся в стеке данных, но это не так. Стек реализуется в виде непрерывной последовательности ячеек памяти, снабженных *указателем стека* (SP); тогда выборку можно представить как копирование элемента, на который указывает SP, а затем увеличение SP на 1 с тем, чтобы он указывал на новый верхний элемент стека. Операция занесения в стек выполняет обратную последовательность действий.

Теперь сложение можно описать следующим образом:

```
ALU ← (SP),   SP  ← SP+1,        ALU ← operand 1, increment SP
ALU ← (SP),   SP  ← SP+1,        ALU ← operand 2, increment SP
ADD                              zero address ADD instruction
SP  ← SP-1, (SP) ← result,       top-of-stack ← sum from ALU
```

где SP — адрес вершины стека, а (SP) — содержимое элемента памяти с этим адресом (TOS). На рис. 2.8 иллюстрируется выполнение команды сложения, два элемента стека заменяются одним — результатом.

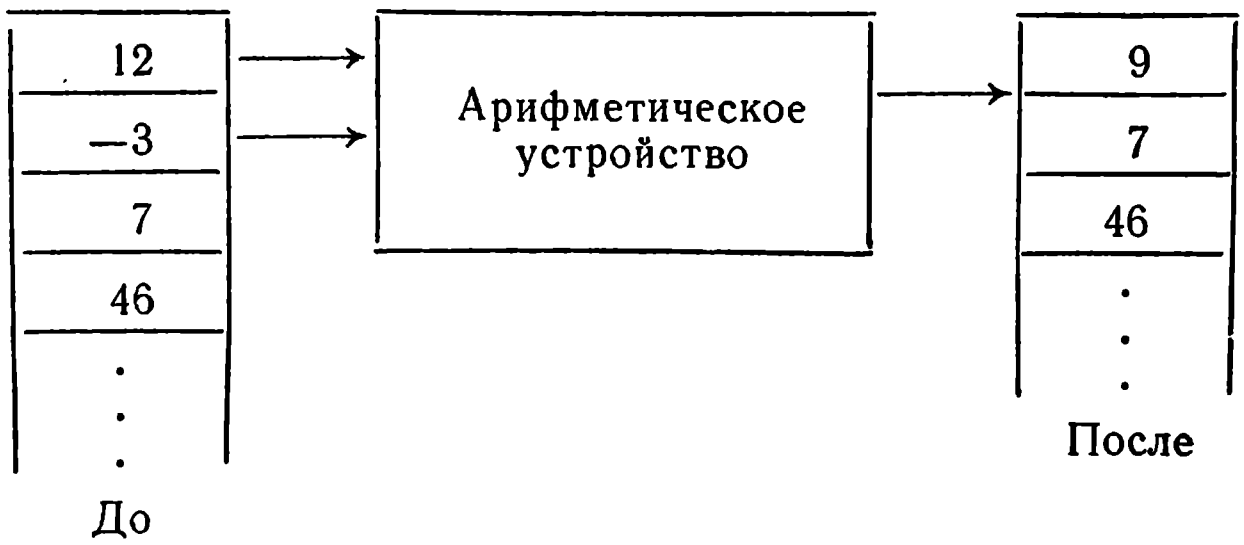


Рис. 2.8. Стековые арифметические операции

Как же теперь можно представить арифметическое выражение

```
A = (B * C) - (D * E)  ?
```

Возможна такая последовательность кодов:

```
PUSH D
PUSH E
MUL
PUSH B
PUSH C
MUL
SUB
POP  A
```

Очевидно, что такая ЭВМ не совсем безадресная, команды чтения и записи из стека требуют по одному адресу памяти, они подобны командам LAC и DAC одноадресной машины. Таким образом, машина, которую мы называли безадресной, имеет только некоторые безадресные команды в своем наборе команд. Например, Burroughs 5500 — стековая машина, имеет как безадресные команды арифметических операций, так и одноадресные команды загрузки в стек и записи в память. Другой пример — это машины PDP-11/40 и PDP-11/03 (известная, как LSI-11), в дополнение к командам сложения, вычитания, умножения и деления, имеют набор команд с плавающей запятой (FIS), оперирующих элементами стека.

2.4. ОПИСАНИЕ И ОБРАБОТКА АДРЕСОВ

Исходя из того, что при любой архитектуре ЭВМ в команде требуется явно указать один или более адресов, можно поставить несколько вопросов. В частности: каким образом можно задавать адреса? как ими можно манипулировать? Для того чтобы на

простых примерах ответить на эти вопросы, введем небольшой набор команд машины двухадресного типа.

Операция	*Символическая форма*	*Значение*
Move	MOV A,B	(B) ← (A)
Add	ADD A,B	(B) ← (B)+(A)
Subtract	SUB A,B	(B) ← (B)-(A)
Compare	CMP A,B	evaluate (A)-(B)
Branch (= 0)	BEQ X	IF LAST OPERATION | = 0 | Then (PC) ← X
Branch (≠ 0)	BNE X	IF LAST OPERATION | ≠ 0 | Then (PC) ← X
Branch (≥ 0)	BPL X	IF LAST OPERATION | ≥ 0 | Then (PC) ← X
Branch (< 0)	BMI X	IF LAST OPERATION | < 0 | Then (PC) ← X
Branch always	BR X	(PC) ← X
Halt	HALT	machine stops

Этот набор в основном совпадает с соответствующими командами PDP-11, но несколько отличается по *внутреннему представлению* и выполняемым *действиям*. В гл. 3 при рассмотрении системы PDP-11 мы обратим внимание на эти различия.

В этом разделе мы воспользуемся символическими адресами операндов, хотя более детально они рассмотрены в параграфе 2.6.3. Запись вида «X:» описывает символический адрес, который присваивается команде или элементу данных, следующим за ней в той же строке, поэтому к этой строке возможно обращение по символическому адресу «X». Например, в параграфе 2.3.1 мы резервировали символические адреса I1, I2, I3, и I4 для четырех последовательных ячеек памяти в неявной форме. Однако на практике символические имена обычно присваиваются тем ячейкам памяти, на которые имеются ссылки в командах программы. В рассматриваемых ниже примерах символические адреса присваиваются лишь первому элементу блока данных, расположенных последовательно. Кроме того, для лучшего понимания программы в тексте имеются комментарии, которые отделяются от команд знаком «;».

2.4.1. Основы описания адресов

Исторически сложилось так, что развитие технических средств определяется требованиями программного обеспечения. Программисты определили, что необходимо иметь разнообразные методы адресации, а затем эти требования нашли свое отражение в технических средствах. На примере простой программы на языке высокого уровня мы рассмотрим, чем были вызваны различные методы описания адресов. Суммирование элементов массива представляет собой одну из наиболее распространенных операций при работе с массивами. Приведенный ниже пример написан на Фортране, такое решение типично для большинства языков высокого уровня.

```
      .
      .
      ISUM = 0
      DO 10 I = 1,50
[10   ISUM = ISUM + K(I)]
```

Эта же задача на языке Ассемблера двухадресной машины может быть решена следующим образом:

```
          .
          .
          MOV     ZERO,ISUM          ;INITIALIZE SUM
          ADD     K,ISUM             ;ADD FIRST K VALUE
          ADD     K+1,ISUM           ;ADD SECOND K VALUE
          ADD     K+2,ISUM           ;ADD THIRD K VALUE
          .
          .
          ADD     K+49,ISUM        ;NOTE: DECIMAL 49
          .
          .
ZERO:     0
ISUM:     0
K:     (ARRAY OF 50 ELEMENTS STARTS HERE)
```

Решение предполагает, что массив K занимает последовательные слова в памяти и что программа построена как линейная последовательность шагов. Допустим, что суммирующая часть программы начинается с адреса 1000, начальный адрес массива K — 2000, ISUM хранится в ячейке с адресом 3000. Допустим также, что код команды сложения будет 10; тогда линейная последовательность команд в том виде, в котором она хранится в памяти, для выполнения будет выглядеть так:

Ячейка	*Содержимое*
1000	10 2000 3000
1001	10 2001 3000
1002	10 2002 3000

. . . .

В рассматриваемой нами гипотетической машине *оба* адреса хранятся в одном слове вместе с кодом операции.

Этот пример хорошо иллюстрирует то, что программисты называют «синдромом плюс или минус единица». Если спросить, сколько чисел между 19 и 27, то обычно можно получить ошибочный ответ — «8». Совершенно ясно, что программист должен помнить о числах на границах отрезка числовой оси. В данном случае имеется аналогичная трудность: какой адрес у последнего элемента — K + 49 или K + 50? Решение в приведенном примере выполнено совершенно верно, следует помнить, что если первый адрес массива K + 0, то последний должен быть на 1 меньше, чем число его элементов. С помощью модификации адреса можно получить решение, более эффективное с точки зрения использования адресного пространства, но менее эффективное по времени.

2.4.2. Модификация адреса

Любому программисту известно о применении в программе *циклов*. Их организация является одним из самых мощных инструментов, которыми пользуется программист, и позволяет ему многократно повторять одни и те же команды, выполняя при этом аналогичные действия. Использование циклов придает программе

гибкость, поскольку, меняя в зависимости от условий начальное значение счетчика циклов, можно варьировать число повторений некоторой последовательности команд.

В данном случае в рассматриваемой программе (рис. 2.9 б) изменение условий — это изменение адресов. Каждая команда сложения ADD подобна предыдущей, за исключением того, что адрес операнда на единицу больше. Следовательно, если мы хотим заменить линейную часть программы циклической, то в каждом повторении цикла мы должны модифицировать адресную часть команды ADD.

Мы уже говорили о концепции фон Неймана. Одно из следствий этой концепции состоит в том, что машина не делает различия между данными и командой. В нашем примере это означает, что одна программа может изменять другие, а также и самое себя, производя таким образом новые программы. В частности, в программе могут быть использованы команды, модифицирующие операнд или адресную часть команд обращения в память.

Приме́ним модификацию адреса к задаче о массиве для нашей гипотетической двухадресной машины. Единственная разница между командами состоит в значении первого адреса. Будет логично взять содержимое ячейки 1001, прибавить 1 к первому адресу и получить в результате содержимое ячейки 1002, например:

```
    contents of location 1001,      10 2001 3000
                                        +1
 =  contents of location 1002,      10 2002 3000
```

На рис. 2.9а приведена желаемая блок-схема.

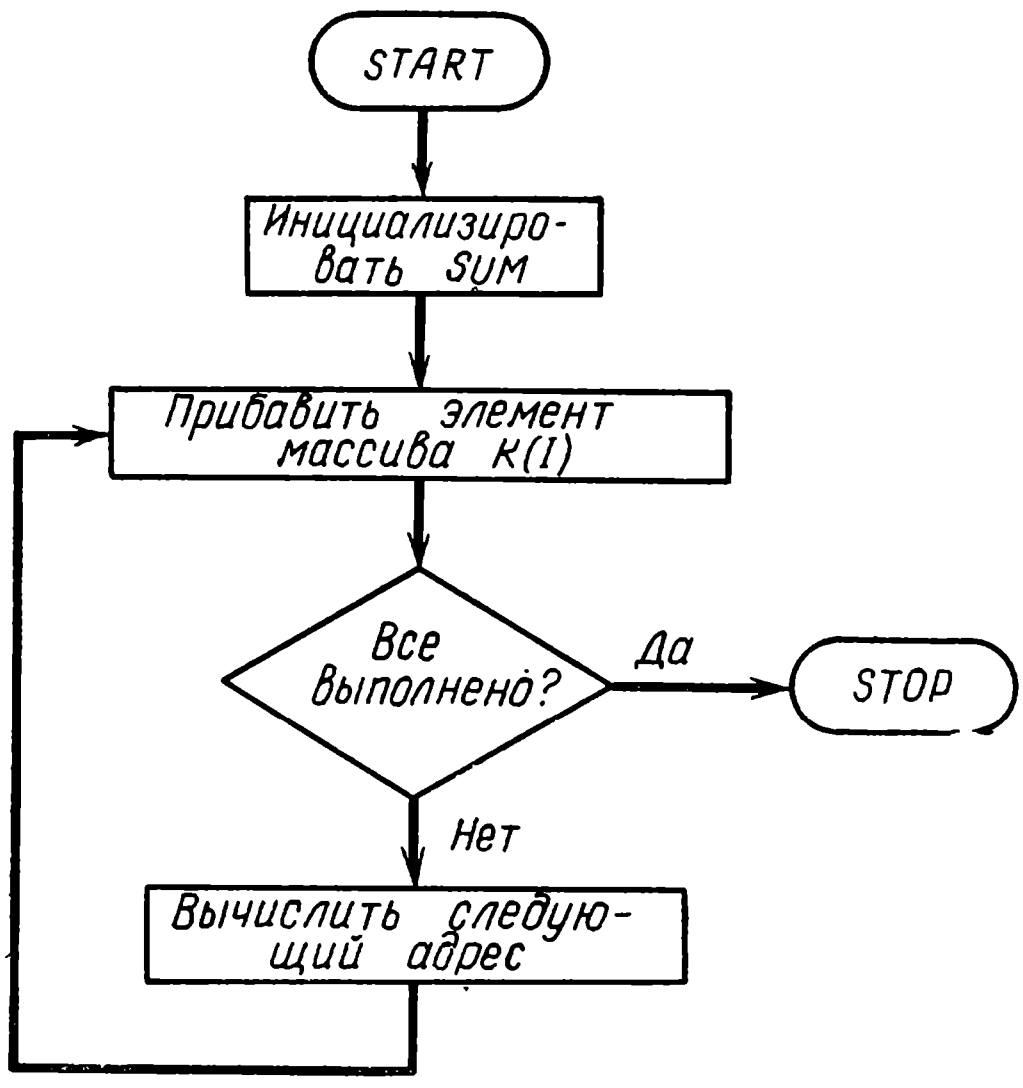


Рис. 2.9а. Суммирование элементов массива K

В результате получим следующий сегмент программы:

```
          .
          .
START:    MOV      ZERO,ISUM          ;INITIALIZE SUM
FIXUP:    ADD      K,ISUM             ;ADD ELEMENT OF ARRAY K
          CMP      FIXUP,LIMIT        ;DONE YET?
          BEQ      STOP               ;YES
          ADD      CONS,FIXUP         ;MODIFY ADDRESS
          BR       FIXUP              ;LOOP
STOP:     HALT
ZERO:     0                           ;CONSTANT ZERO
ISUM:     0                           ;RESULTANT SUM
CONS:     (A CONSTANT)                ;A SUITABLE VALUE SO AS TO
                                      ;INCREMENT THE SOURCE OPERAND
                                      ;ADDRESS OF "ADD K,ISUM" BY 1
LIMIT:    ADD      K+49,ISUM          ;SYMBOLIC REPRESENTATION OF "ADD"
                                      ;INSTRUCTION AFTER FINAL LOOP
K:        (ARRAY OF 50 ELEMENTS STARTS HERE)
```

Рис. 2.9б. Программа, модифицирующая себя

В этом примере читатель может заметить несколько интересных моментов. Во-первых, команда с меткой FIXUP модифицируется путем прибавления единицы к адресу первого операнда. Таким образом формируется новый адрес внутри команды, и при последующем выполнении будет выбираться очередной элемент массива, который будет суммироваться с ISUM. Во-вторых, слово с меткой LIMIT содержит значение, которое должна принять команда FIXUP, когда будет просуммирован последний элемент массива K. При этом команда CMP сравнивает значение двух команд, которые на самом деле являются двумя числовыми значениями. В-третьих, эта программа сама себя модифицирует, и она не является повторно входимой, т. е. по окончании ее выполнения команда FIXUP имеет модифицированное значение адреса первого операнда, следствием чего будет получение ошибочных результатов при попытке повторного выполнения программы.

Хотя как средство программирования модификация адресов и команд может оказаться полезной, следует избегать этого приема. Некоторые объясняющие это причины таковы:

1) программы с модификацией очень трудны в отладке;

2) в современных вычислительных системах требуется неизменность кодов программ в связи с тем, что в них имеются такие средства, как защита, распределение программ между различными пользователями, кроме того, неизменные программы более эффективно управляемы. Имеется и ряд других причин.

Избежать модификации команд можно с помощью некоторых иных средств нашей гипотетической машины.

2.4.3. Использование регистров общего назначения

Система команд машины наиболее эффективна в том случае, если в ее составе имеются «дополнительные» команды, которые в некоторой степени дублируют свойства основных. Их применение облегчает программирование и повышает эффективность про-

грамм по быстродействию и по использованию памяти. Значительное увеличение скорости работы программ и снижение стоимости программного обеспечения также может быть достигнуто, если при проектировании машины предусматриваются средства для работы с простыми элементами данных, например, такими, как одномерные массивы или *векторы*. В частности, употребление нескольких *регистров общего назначения* позволяет программисту применять эффективные по времени методы адресации операндов. Приведенный выше пример суммирования элементов массива хорошо подходит для иллюстрации использования регистров общего назначения. Они могут служить в качестве указателей адресов или *индексными регистрами*, а также в качестве аккумуляторов данных.

Если регистр используется как индексный, то *исполнительный* адрес операндов источника или приемника вычисляется суммированием значений операнда или индексного слова (X) и указанного индексного регистра (R). Значением операнда может быть число со знаком. В результате эффективный исполнительный адрес (EA) вычисляется в форме

$$X + (R) = EA,$$

где X может быть положительным или отрицательным числом. Индексированная команда сложения, например, может быть записана в следующем символическом виде:

```
ADD  X(R),ISUM.
```

Допустим, что наша гипотетическая машина имеет два регистра общего назначения — R0 и R1. Каждый из них может быть использован для хранения адреса, слова данных или для индексации. Обычно регистры общего назначения представляют собой составную часть процессора. Они имеют аппаратурную реализацию и, следовательно, обладают высоким быстродействием, поэтому их можно эффективно применять для хранения часто используемых переменных. В нашей машине регистры могут служить в качестве аккумуляторов и индексных регистров.

Рассмотренный выше пример программы может быть переписан с использованием регистров общего назначения. Хотя полученный код (рис. 2.10) длиннее, чем в предыдущей модифицирующей себя программе, он не модифицируется и инициализируется автоматически, т. е. R0 и R1 обнуляются при каждом выполнении.

В данном примере R0 используется как аккумулятор, а R1 — как индексный регистр. После обнуления обоих регистров программа выполняет суммирование элементов массива K, формируя исполнительный адрес первого операнда из содержимого R1 и базового адреса K. Значения исполнительного адреса можно представить следующей последовательностью: K, K + 1, K + 2, ..., K + 49. Команда CMP, сравнивая R1 с десятичным значением 49, ограничивает выполнение цикла. До тех пор, пока не будет R1 < 50, содержимое увеличивается на 1 и цикл повторяется. Как

только содержимое R1 достигнет значения 50, программа завершится. Внимательный читатель заметит, что при использовании индексной адресации сравниваемые значения являются уже не адресами, а значениями смещения относительно некоторой базы.

```
          .
          .
          MOV     ZERO,R0          ;SUM IN REGISTER 0
          MOV     ZERO,R1          ;COUNT IN REGISTER 1
LOOP:     ADD     K(R1),R0         ;ADD IN ELEMENT OF ARRAY K
          CMP     R1,D49           ;DONE?
          BPL     STOP             ;YES!
          ADD     ONE,R1           ;NO, GENERATE NEXT INDEX
          BR      LOOP             ;CONTINUE LOOP
          .
          .
STOP:     HALT                     ;STOP HERE
ZERO:     0                        ;CONSTANT ZERO
ONE:      1                        ;CONSTANT ONE
D49:      49                       ;CONSTANT 49
K:        (ARRAY OF 50 ELEMENTS)
```

Рис. 2.10. Индексная адресация

2.4.4. Непосредственный метод

Сделаем небольшое отступление, прежде чем продолжить дальнейшее обсуждение методов адресации. Это отступление относится к употреблению символических имен для часто применяемых в программе констант. Например, команда, использующая константу 2, может иметь следующий вид:

```
ADD  TWO,N
```

где константа хранится в ячейке памяти с символическим адресом «TWO». Более удобна иная форма задания числа, суммируемого с N, не требующая явного задания адреса слова, хранящего значение константы; она включает константу в команду как составную часть. Такой метод адресации называется *непосредственным.* Используя его, можно, например, записать сложение 2 и N так:

```
ADD     #2,N
```

Здесь непосредственный режим задается предшествованием числу или имени знака «#». Таким образом, вместо того, чтобы использовать в команде *адрес* операнда, возможно включение в нее его значения.

Если учесть, что символ «#» служит для обозначения значения, то запись «#n» (где n — число) обозначает константу, а запись «#X» (где X — символический адрес) — адрес, на который указывает X. Следует отметить, что величина непосредственного значения ограничена размером замещаемого им адресного поля, т. е. она не может быть больше, чем значение адреса.

Проиллюстрируем использование непосредственного метода адресации на примере уже известной задачи о массиве; теперь эта

задача может быть запрограммирована так:

```
        .
        MOV     #0,R0           ;R0 ← 0
        MOV     #49,R1          ;R1 ← 49
LOOP:   ADD     K(R1),R0        ;ADD IN ELEMENT OF ARRAY K
        SUB     #1,R1           ;R1 ← R1 -1
        BPL     LOOP            ;LOOP WHILE R1 ≥ 0
        .
        .
```

Общее число операторов в этом случае уменьшилось. Однако это уменьшение произошло не вследствие применения непосредственного метода адресации, а как результат нового приема программирования. Он использует свойство команды вычитания SUB, которая информирует устройство управления о результате последней арифметической операции точно так же, как это делается в команде CMP. Тогда, суммируя числа в обратном порядке от последнего к первому и изменяя содержимое R1 в сторону уменьшения, а не возрастания, можно избавиться от применения команды CMP. Вместо нее после каждого вычитания анализируется значение R1 и если результат не отрицательный, то цикл продолжается.

2.4.5. Автоиндексация

Индексация используется в программировании настолько часто, что в некоторых машинах имеется специально предназначенный для этой цели аппарат, автоматически уменьшающий или увеличивающий значения индексных регистров.

В нашей машине имеются *автоинкрементный* и *автодекрементный* методы адресации, обеспечивающие автоматическое пошаговое изменение индексных регистров для обращения к элементам в таблицах. Этот метод предполагает, что содержимое соответствующего общего регистра является адресом операнда или указателем на операнд. Поскольку указатель употребляется для последовательности адресов, то он всегда адресует очередной элемент таблицы.

В автодекрементном методе, который записывается в форме —(R), содержимое регистра всегда уменьшается *прежде*, чем его используют в качестве адреса операнда. В автоинкрементном методе, записываемом в форме (R)+, содержимое регистра меняется сразу же *после* того, как оно было употреблено в качестве адреса операнда.

Возвращаясь опять к задаче о массиве, можно теперь переписать ее так:

```
        .
        .
        MOV     #0,R0           ;R0 ← 0
        MOV     #K,R1           ;R1 ← START ADDRESS OF ARRAY K
LOOP:   ADD     (R1)+,R0        ;R0 ← R0 + NEXT ELEMENT OF K
        CMP     #K+49,R1        ;DONE?
        BPL     LOOP            ;NO!
```

Это одно из наиболее коротких возможных решений.

2.4.6. Косвенная адресация

Косвенная адресация является альтернативой индексной. В некоторых машинах реализован только один из этих методов адресации, но в нашей гипотетической ЭВМ заложены оба. Использование *косвенного метода* в сочетании с другими, такими, как регистровый, индексный или автоинкрементный, дает возможность получить весьма изощренные приемы адресации.

Существует множество способов применения косвенной адресации. Например, с ее помощью можно обеспечить передачу параметров в подпрограмму (как альтернатива индексному методу); кроме того, косвенную адресацию можно использовать для создания структур данных и для многого другого. Все это осуществимо потому, что при косвенной адресации задается не просто «адрес операнда», а «адрес адреса операнда».

В ассемблере для задания косвенной адресации служит знак «@» [или, если при этом не возникает двусмысленности, можно употребить ( )]. В качестве примера возьмем обнуление ячейки HERE:

```
        MOV     #0,@INDR        ;CLEAR OPERAND SPECIFIED BY
                                ;       CONTENTS OF "INDR"
        .
INDR:   HERE                    ;CONTAINS ADDRESS OF "HERE"
        .
        .
HERE:   77                      ;"77" OVERWRITTEN BY "0"
```

Те же методы, которые применялись для обнуления ячейки, могут быть использованы и для обнуления массива. На рис. 2.11 приведены пример программы обнуления массива и соответствующая ей блок-схема. В этом примере командой MOV в R1 заносится адрес массива A (поскольку первый операнд — это #A, то адрес A заносится в R1); теперь, с помощью содержимого R1, можно адресоваться косвенным образом.

В обоих приведенных примерах осуществляется косвенная адресация с глубиной вложения, равной единице. Машина позволяет обеспечить большую глубину вложения, адресуясь к ячейке, которая, в свою очередь, содержит адрес другой ячейки, к которой выполняется косвенная адресация. Многоуровневая адресация требует, чтобы в ячейке был признак того, прямо или косвенно используется содержащийся в ней адрес.

Выбор между одно- и многоуровневым методами адресации определяется, с одной стороны, их удобством, с другой — стоимостью. В распространенных теперь мини-ЭВМ, характеризующихся невысокой стоимостью, машинное слово короткое и каждый его бит должен быть использован с максимальной эффективностью, поэтому в них обычно не реализуется многоуровневая косвенная адресация. По экономическим соображениям максимально возможное число бит в команде применяется для прямой адре-

сации памяти, вследствие этого бит признака косвенной адресации употребляется как часть команд, а не как часть адреса операнда.

Для тех ЭВМ, в которых имеются оба метода адресации, необходимо рассмотреть вопрос о том, команды с каким из способов адресации следует выполнять прежде — с индексной или косвенной? Хотя на первый взгляд может показаться, что ответ на этот

```
START:  MOV     #100,R0     ;LOOP COUNT

        MOV     #A,R1       ;ARRAY BASE ADDRESS

LOOP:   MOV     #0,(R1)+    ;CLEAR "@R1",
                            ;THEN INCREMENT R1

        SUB     #1,R0       ;DECREMENT LOOP COUNT

        BNE     LOOP        ;LOOP IF R0 > 0

        HALT                ;STOP

A:      (ARRAY OF 100 ELEMENTS STARTS HERE)
```

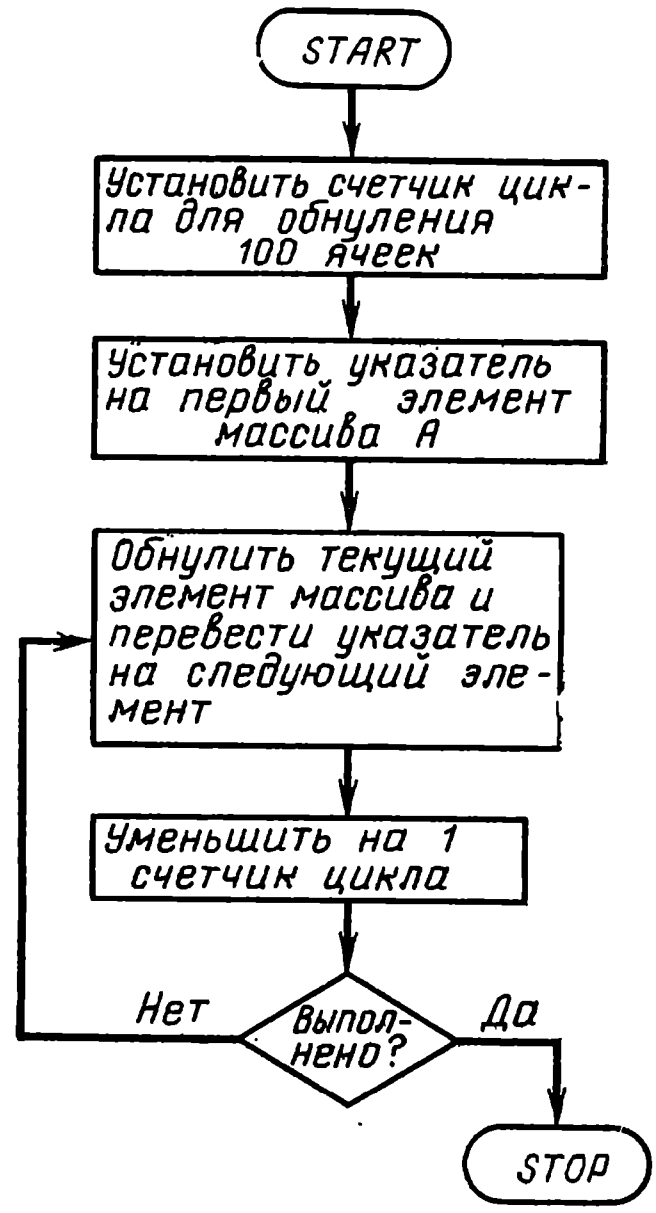


Рис. 2.11. Использование косвенной адресации

вопрос не имеет существенного значения, иногда в силу определенных причин предпочтительнее раньше выполнить индексную, а затем косвенную адресацию или наоборот.

Когда машина имеет короткое слово и невозможна прямая адресация ко всем ячейкам памяти, тогда в первую очередь следует выполнить косвенную адресацию. Это объясняется тем, что обычно используемое в качестве косвенного адреса слово бывает длиннее, чем адресное поле команды, и поэтому с помощью косвенной адресации можно обратиться в любую область памяти, а затем уже выполнять индексирование. Если же длина слова позволяет адресовать всю память, то предпочтительнее прежде исполнять индексацию, поскольку это облегчает, например, передачу параметров в подпрограмму.

2.5. ЭТАПЫ ВЫПОЛНЕНИЯ КОМАНД

В предыдущих разделах наше внимание было сконцентрировано на адресации операндов. Для более полного понимания ряда

вопросов теперь необходимо рассмотреть, как выполняются отдельные команды. На рис. 2.12 схематически показаны отдельные этапы этого процесса.

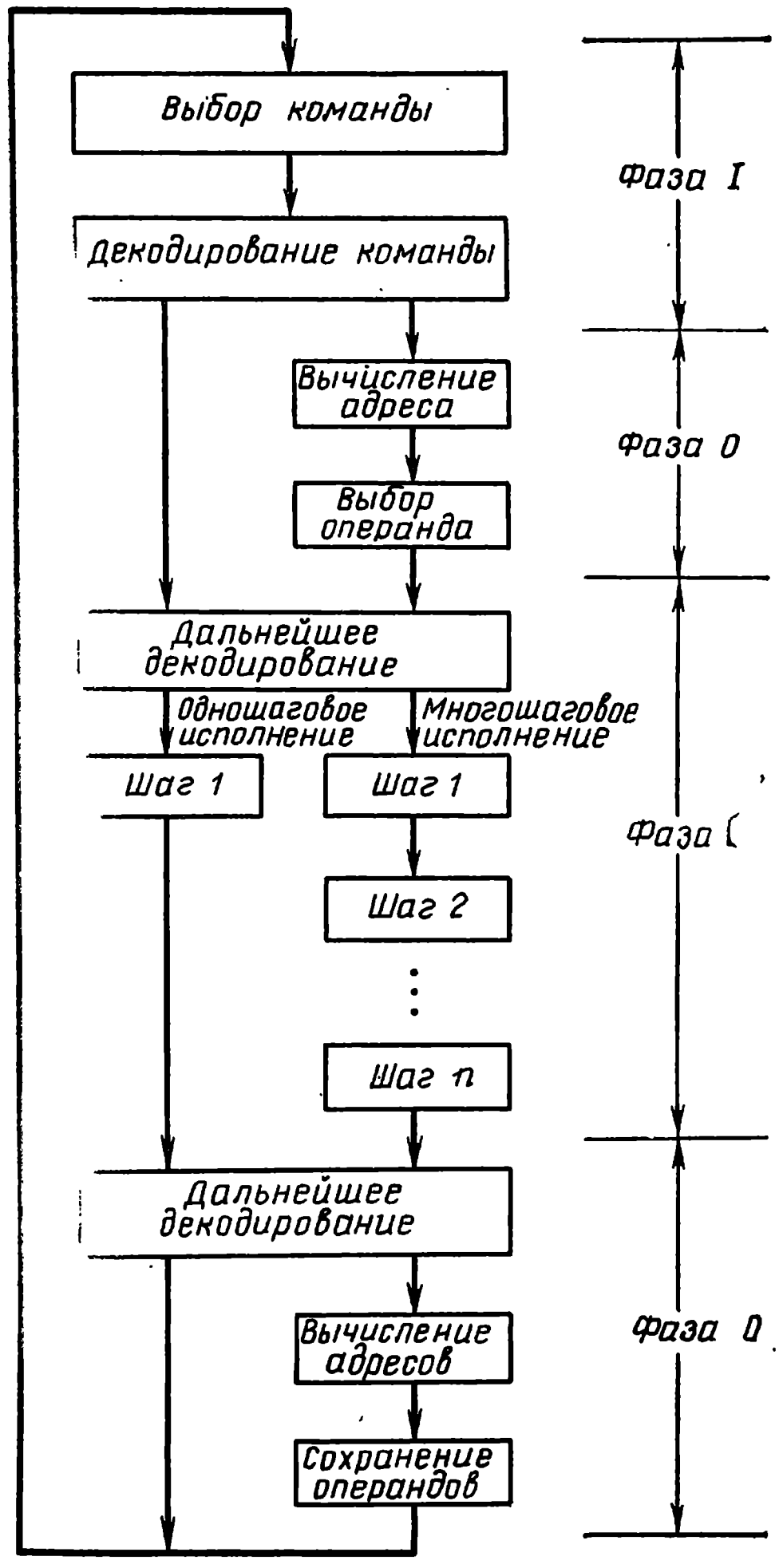


Рис. 2.12. Этапы выполнения команд в ЭВМ

2.5.1. Команда, операнд и фазы выполнения

Вначале команда выбирается из памяти и декодируется (*I-фаза*). Затем вычисляется адрес или адреса одного или нескольких операндов, которые выбираются из памяти (фаза операндов, или *O-фаза*). Сложность *O*-фазы зависит от методов адресации. На следующей, *E-фазе* выполняется заданная операция над операндами. Для разных команд объем действий, производимых

в этой фазе, различен. Например, команда пересылки MOV не требует арифметических операций и использует два операнда. В то же время двухоперандная команда умножения MUL вовлекает в исполнение несколько арифметических операций. В том случае, если необходимо сохранить результат операции, привлекается дополнительная *O*-фаза.

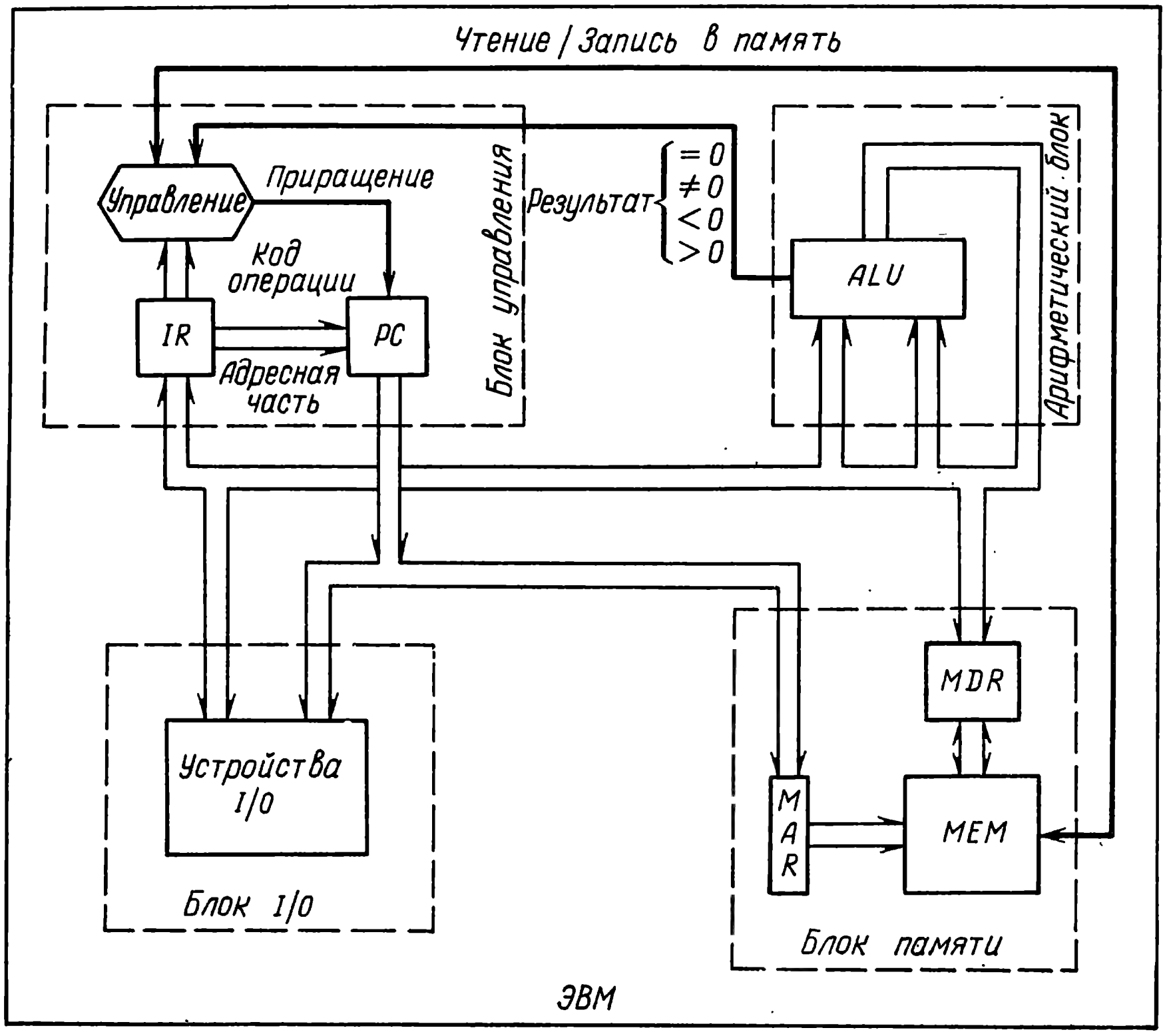


Рис. 2.13. Упрощенная диаграмма передачи данных в ЭВМ

Сложность этапов выполнения команды, а следовательно, и время выполнения определяются целым рядом факторов. К примеру, команда MUL — умножение с фиксированной запятой — реализуется как последовательность сложений и сдвигов, т. е. ее *E*-фаза повторяется многократно. В тех случаях, когда команды умножения MUL в системе команд машины отсутствуют, умножение осуществляется в виде последовательности команд сложения и сдвига. Очевидно, что предпочтительнее многократное выполнение *E*-фазы внутри команды. Точно так же эффективнее реализация косвенной адресации путем многократного выполнения *O*-фазы, чем в виде последовательности команд.

Какими же ресурсами должна обладать машина для того, чтобы она могла выполнять команды? При более детальном анализе

структуры ЭВМ, показанной на рис. 2.1, мы обнаружим (см. рис. 2.13), что в машине в дополнение к счетчику команд имеется целый ряд регистров, недоступных программисту:

1. *Регистр команды* (IR) хранит копию выполняемой команды.
2. *Регистр адреса памяти* (MAR) хранит адрес памяти.
3. *Регистр данных в памяти* (MDP) используется как буфер для хранения данных, пересылаемых между памятью и CPU.

С учетом этого выполнение команды сложения «ADD X, Y» может быть описано так:

```
I-фаза
------
1. (MAR) ← (PC)                    ;ADDRESS OF INSTRUCTION INTO MAR
2. (MDR) ← ((MAR))                 ;FETCH INSTRUCTION
3. (PC)  ← (PC) + 1                ;INCREMENT PC
4. (IR)  ← (MDR)                   ;COPY INSTRUCTION INTO IR
5. DECODE INSTRUCTION

O-фаза
------
1. (MAR) ← (IR address opnd1)      ;COMPUTE ADDRESS OF OPERAND1
2. (MDR) ← ((MAR))                 ;FETCH OPERAND1
3. (Rx)  ← (MDR)                   ;ROUTE OPERAND1 TO ALU RIGHT PORT
4. (MAR) ← (IR address opnd2)      ;COMPUTE ADDRESS OF OPERAND2
5. (MDR) ← ((MAR))                 ;FETCH OPERAND2
6. (Ry)  ← (MDR)                   ;ROUTE OPERAND2 TO ALU LEFT PORT

E-фаза
------
1. (MDR) ← (Rx) + (Ry)             ;ADD

O-фаза
------
1. ((MAR)) ← (MDR)                 :STORE RESULT
```

Здесь Rx и Ry — регистры ALU. В параграфе 3.5.4 мы рассмотрим исполнение фаз *I*, *O* и *E* в ЭВМ PDP-11. Далее, в гл. 6, мы рассмотрим внутренние архитектурные компоненты, применяемые для реализации системы команд в машине PDP-11/60.

2.6. СИМВОЛИЧЕСКИЙ АССЕМБЛЕР, СРЕДСТВА ПРОГРАММИРОВАНИЯ

Кодирование программ на машинном языке в цифровой форме представляет собой довольно трудную задачу. От программиста требуется знание кодов операций и адресов операндов в виде чисел. Из рассмотренных в предыдущих разделах примеров мы видим, что много проще программировать и читать написанные ранее программы с использованием мнемоник — вспомогательных, служащих для запоминания *кодов операций* и *операндов имен.* Мнемоники позволяют присваивать операндам символические имена, например SUMX, ABS, COUNT и т. п., подобно тому, как

это делается в языках высокого уровня. Отсюда следует, что *символический Ассемблер* является важным средством системы программирования.

2.6.1. Ассемблирование

Ассемблированием называют процесс перевода программы с языка символического кодирования в машинный язык. Этот процесс легко поддается автоматизации и может выполняться самой машиной. Символический Ассемблер — это переводящая программа, функционально подобная компилятору с Фортрана, трансформирующая программу, написанную на *исходном* языке, в другую, уже на *объектном* языке (см. рис. 2.14). Ассемблер отличается от компилятора, во-первых, сложностью исходного языка, поскольку он обычно выполняет перевод «один-в-один» символических исходных операторов в машинный язык. Компилятор же, обрабатывающий операторы языка высокого уровня, ставит в соответствие одному оператору большое количество машинных команд. Еще одно отличие состоит в том, что выходные объекты компилятора могут быть представлены не на машинном языке, а на некотором промежуточном, например на языке Ассемблера, и должны пройти дальнейшие шаги трансляции.

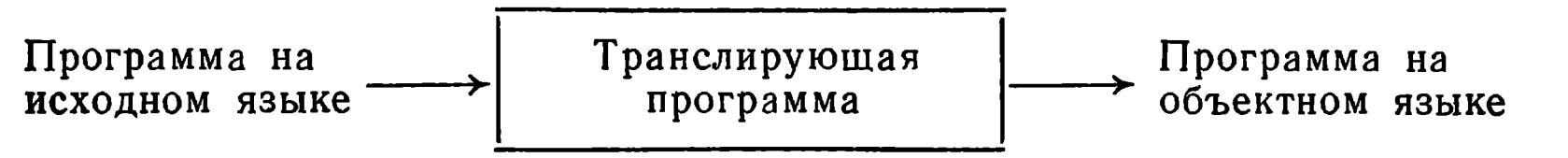


Рис. 2.14. Трансляция программы

Ассемблирование символической программы осуществляется на основе следующего:

1. Просмотр последовательности команд и резервирование памяти для них.
2. Просмотр используемых идентификаторов и резервирование для них памяти.
3. Преобразование символических кодов операций и идентификаторов в их машинные эквиваленты.

Для того чтобы реализовать эти действия, ассемблер должен включать счетчик адресов и таблицу идентификаторов.

2.6.2. Счетчик адресов

Большинство символических программ пишется в предположении, что последовательно написанные команды будут располагаться в последовательных ячейках памяти. Если программист первой команде программы присваивает абсолютный адрес, то остальные будут располагаться в последующих ячейках памяти.

Допустим, что ассемблер — некоторая логическая машина, тогда в ней *счетчик адресов* (LC) играет ту же роль, что *счетчик команд* (PC) в реальной ЭВМ. Функция LC состоит в сохранении

в памяти адреса, куда должна быть записана очередная команда. После того как она оттранслирована, значение LC получает приращение, что служит отражением факта занятости текущей ячейки.

2.6.3. Символические адреса

Программист не знает (или не желает знать) абсолютных адресов ячеек памяти, служащих для хранения констант и результатов или адресов, на которые должен быть выполнен переход. Написание программ в этом случае возможно только за счет применения символических адресов — идентификаторов, которым ассемблер присвоит соответствующие числовые значения. При ассемблировании имени присваивается текущее значение счетчика адреса, и идентификатор рассматривается как *метка* или указатель на ячейку, которую эта метка представляет. Для того чтобы учесть присвоенные идентификаторам в процессе ассемблирования значения, ассемблер ведет *таблицу идентификаторов,* в которой эти значения хранятся. Для распознавания того, что идентификатор используется в качестве метки, вводится специальный символ — терминатор. В предыдущих примерах в качестве терминатора употреблялось двоеточие, например «А:».

В целом задача программиста, работающего на языке символического Ассемблера, состоит в написании *исходных программ,* на основании которых ассемблер порождает *объектные* модули на машинном языке. Используя идентификаторы для переменных и меток, программист может писать программы, независимые от абсолютного расположения в памяти. Действительно, символические имена независимы от абсолютных адресов ячеек, поэтому можно произвольным образом вставлять или удалять соответствующие команды.

Итак, ассемблер воспринимает на входе символические команды и на их основании порождает операторы машинного языка по принципу «один-в-один». Символическое программирование в таком случае можно определить как метод, использующий имена для переменных, операций и команд. Однако в процессе программирования нужно задать не только что должно быть обработано ассемблером, но каким образом обработка должна быть выполнена. Для этого служат специальные описательные, исполнительные и управляющие операторы.

Три указанных типа операторов ассемблера могут быть определены следующим образом:

1. *Описательные.* Используются для управления распределением памяти для различных имен, входных, выходных и рабочих областей. Эти операторы не являются командами в подлинном смысле, они служат только для резервирования пространства памяти, а также для определения идентификаторов и присваивания ячейкам содержимого.

2. *Исполнительные* — это собственно машинные команды в символической форме.

3. *Управляющие* — это команды, называемые также директивами; адресованы самому ассемблеру и дают возможность программисту управлять процессом трансляции.

2.6.4. Синтаксис языка Ассемблера

Программирование на языке символического Ассемблера подобно программированию на любом другом языке. Для того чтобы предложение исходного текста было разобрано, оно должно удовлетворять строгим правилам. Поскольку и ассемблер, и компилятор с Фортрана являются трансляторами с языков, нет ничего удивительного в том, что правила построения предложений языка Ассемблера имеют много общего с аналогичными правилами Фортрана.

Имена, или *идентификаторы*, языка Ассемблера представляют собой произвольную последовательность цифр или букв латинского алфавита (A, B, C, ..., Z), начинающуюся с буквы. Мнемоники кодов операций, определенные для данной конкретной машины, представляют собой имена, значения которым присваиваются в соответствии с таблицей постоянных имен. *Идентификаторы, определенные пользователем*, обычно имеют фиксированную максимальную длину или рассматривается только несколько начальных символов, а остальные игнорируются. Примерами идентификаторов, которые задаются пользователем, могут быть упомянутые в предыдущих примерах NUM, START, ONE и т. д.

Число — это любая последовательность цифр. Для удобства программирования в ассемблерах допускаются различные системы счисления, такие, как десятичная, восьмеричная и др. Однако, если в ассемблере по умолчанию применяется какая-то одна система, например восьмеричная, то при использовании числа́ в другой нужно либо указывать это в явном виде операторами управления, такими, как .BASE, либо за числом должен следовать специальный символ, идентифицирующий систему счисления, например десятичная точка (число 98.— десятичное, а число 98 номинально будет восприниматься как восьмеричное). В исходном символическом тексте программы все идентификаторы и числа, вне зависимости от системы счисления, всегда должны иметь двоичный эквивалент для того, чтобы ассемблер мог интерпретировать входную информацию.

Часто в ассемблере имеется набор специальных символов для обеспечения распознавания меток, директив, десятичных значений, комментариев и т. д. Такие наборы включают:

$: = % # @ ( ) ; " ' + - & !

Сюда же входит непечатаемый символ пробела, который не имеет графического изображения, но оказывает влияние на процесс ассемблирования. Специальные символы используются для задания операций над именами и числами в программе. Например, выше было показано употребление двоеточия для ограничения метки.

До сих пор формат операторов демонстрировался на примерах, однако не приводилось строгого определения формата предложений языка Ассемблера. Это предложение включает четыре поля, каждое из которых может занимать определенную позицию или отделяется от предыдущего специальным символом. Оно имеет следующий формат:

МЕТКА ОПЕРАТОР ОПЕРАНД КОММЕНТАРИЙ

Поля метки и комментария не являются обязательными. Поля оператора и операнда связаны между собой, одно из них может быть опущено в зависимости от содержимого другого. Операнд — это часть предложения, над которой выполняется оператор. Операндами могут быть идентификаторы, выражения или числа. Если в предложении имеется несколько операндов, они должны быть разделены между собой специальными символами (обычно это запятая); аналогично комментариям предшествует точка с запятой.

Те ассемблеры, в которых поля специальным образом разделяются, позволяют *свободную запись* в отличие от встречающихся фиксированных форматов записи. В первом случае программист может управлять форматом предложения, используя для этого пробелы. Если пробелы не входят в состав идентификатора, числа или строки символов (об этом ниже), то они не оказывают влияния на процесс ассемблирования. Пробелы можно использовать для того, чтобы сделать программу более читаемой. Например, оператор

```
LABEL:MOV(R0)+,TAG;POP VALUE OFF A STACK
```

с использованием символов форматирования может быть переписан как

```
LABEL:   MOV     (R0)+,TAG      ;POP VALUE OFF A STACK
```

что явно удобнее для чтения.

2.6.5. Директивы управления транслятором

Назначение директив управления состоит в том, чтобы передавать ассемблеру указания о выполнении определенных действий. Они не являются командами машинного языка, хотя по форме подобны мнемоникам команд, т. е. имеют код команды, операнды и допускают метку. Поэтому их называют также *псевдооператорами* и отличают по тому, что коду директивы предшествует специальный символ, например точка.

Наиболее общим для ассемблеров разных ЭВМ является псевдооператор конца .END:

```
.END label
```

Он сообщает ассемблеру, что это последняя транслируемая строка и выполнение создаваемой программы должно начаться с команды, символический адрес которой указан в его поле операнда. Это очевидный эквивалент оператора END в Фортране. Некоторые другие директивы перечислены ниже.

	Формат псевдооперации	*Функция*
label:	.WORD value	Place "value" in a word with symbolic address "label"
label:	.BLKW value	Reserves contiguous block of "value" words starting at symbolic address "label".
	.TITLE title	Gives a title to a program.

УПРАЖНЕНИЯ

1. Переделайте блок-схему на рис. 2.3 для двухадресной машины.

2. Постройте блок-схему для суммирования всех положительных значений массива K из 50 элементов. Результат поместите в ячейку SUM.

3. Некоторые ограничения ассемблера часто могут показаться излишне жесткими. Они вызваны тем, что программист, создавший ассемблер, хотел исключить двусмысленности, стремился к улучшению распределения памяти и др. Попытайтесь объяснить с позиции человека, создающего двухпроходный ассемблер, чем может быть вызвано каждое из перечисленных ниже ограничений:

а) идентификатор должен начинаться с буквы;

б) идентификатор не должен содержать специальных символов или пробелов;

в) идентификатор не может быть использован в программе в качестве метки дважды.

4. Напишите программу суммирования всех положительных чисел, находящихся в ячейках 3001, 3002, 3003, ... Их количество содержится в ячейке 3000 и принадлежит интервалу $0 \leqslant N \leqslant 99$. Если $N = 0$, то результат должен быть равен 0. Сумма должна храниться в ячейке 2777.

5. Напишите программу, которая эквивалентна приведенной ниже программе на Фортране для массивов A, B и C:

```
      DO 10 I=1,1000
10    C(I)= A(I)+B(I)
```

6. Напишите программу сортировки N чисел, которая не будет использовать дополнительные массивы. Покажите, что она будет работать при всех значениях N в пределах $0 \leqslant N \leqslant 99$. Ваше решение должно быть основано на применении индексных регистров, и полученная программа не должна модифицировать себя.

7. Допустим, что существует ЭВМ с памятью 4096 слов, с адресами от 0 до 4095 и с одним индексным регистром. Исполнительный адрес в команде с индексацией, с индексом X, вычисляется по формуле $EA = X + (R)$. Каково значение величины X, если $R = 0$, а $EA = 4095$.

8. Для той же ЭВМ, что и в примере 7, определите X, если $R = 4095$, а $EA = 0$.

9. Перепишите сегмент программы, приведенной в параграфе 2.4.5, используя вместо автоинкрементной автодекрементную адресацию.

10. Если ЭВМ имеет и индексирование, и косвенную адресацию, то важно знать, что́ из них выполняется раньше в команде, использующей и тот и другой вид адресации. Объясните, почему порядок так важен, на примере двух машин с коротким (допустим, 12 бит) и длинным (допустим, 36 бит) словами.

11. Что означает в машине, имеющей непосредственный и индексный методы адресации, ссылка с одновременным применением обоих методов?

12. Что по вашему мнению предпочтительнее и почему: наличие косвенной адресации с глубиной вложения, равной единице, или наличие индексной адресации?

13. Допустим, что мы имеем двухадресную машину с фиксированной длиной слова, оперирующую десятичными числами со знаком и имеющую полные адресные возможности для обоих операндов, т. е. непосредственную, прямую и косвенную адресацию. В формате символической записи идентификатор, сопровождающийся знаком $, означает непосредственную адресацию, символ # — косвенную адресацию, отсутствие знака — прямую адресацию. Допустим, что знак * обозначает счетчик адресов и является адресом текущей ассемблируемой команды. В системе команд имеются следующие (здесь же приведена их интерпретация):

```
ADD   A1,A2    (A1) ← (A1) + (A2)
SUB   A1,A2    (A1) ← (A1) - (A2)
BGT   A1,A2    if (A1) ≥ (A2) skip next instruction
BZE   A1,A2    if (A1) = 0 jump to A2
```

где A1 и A2 — адресные выражения; в них может быть использован любой из доступных методов адресации. Допустим также, что переменные имеют следующие адреса и значения:

Идентификатор	*Ячейка памяти*	*Содержимое*
A	0	2
B	1	1
	2	0
	3	-1
	4	3
C	5	4
D	6	5

Рассмотрите, на содержимое каких ячеек повлияет выполнение каждой из приведенных ниже последовательностей команд. Например, исполнение команды ADD A, B изменяет содержимое ячеек 0 и 3. Для каждого из примеров предполагаются одни и те же начальные условия.

а.
```
SUB A,A
```

б.
```
ADD B,5$
```

в.
```
SUB B+2,C#
```

г.
```
BGT A$,B#
ADD A#,B+3#
```

д.
```
BZE 2#,*+2
SUB D#,B+1$
```

БИБЛИОГРАФИЯ

В этой главе ставилась задача — изложить основные принципы программирования вне связи с определенным типом ЭВМ. Такой же подход использован в работах Abrams M. D. и Stein P. G. (1973), Walker T. M. (1972), Flores I. (1966) и Mauer W. D. (1968). Будучи в целом достаточно общими, книги Stone H. S. (1975), Gear C. W. (1974) и Katzan H. (1971) ориентированы на определенные модели машин. Стиль изложения материала в этих пособиях позволяет использовать их в учебных целях совместно со справочными руководствами, относящимися к тому типу ЭВМ, который интересует читателя. Гл. 3 полностью посвящена машине PDP-11. Тем читателям, которые хотят более глубоко изучить эту машину, следует обратиться к «PDP-11 Processor Handbook». Более фундаментальные вопросы архитектуры вычислительных машин рассматриваются в работах Stone H. S. (1975), Tannenbaum A. C. (1976), Hayes J. P. (1978).

Глава 3

ЭВМ PDP-11

Семейство PDP-11 состоит из высокоскоростных, универсальных цифровых ЭВМ, оперирующих в основном с 8- или 16-битовыми двоичными числами. На машинах реализуются нуль-, одно- и двухадресные команды, использующие восемь общих регистров для спецификации операндов при различных режимах адресации.

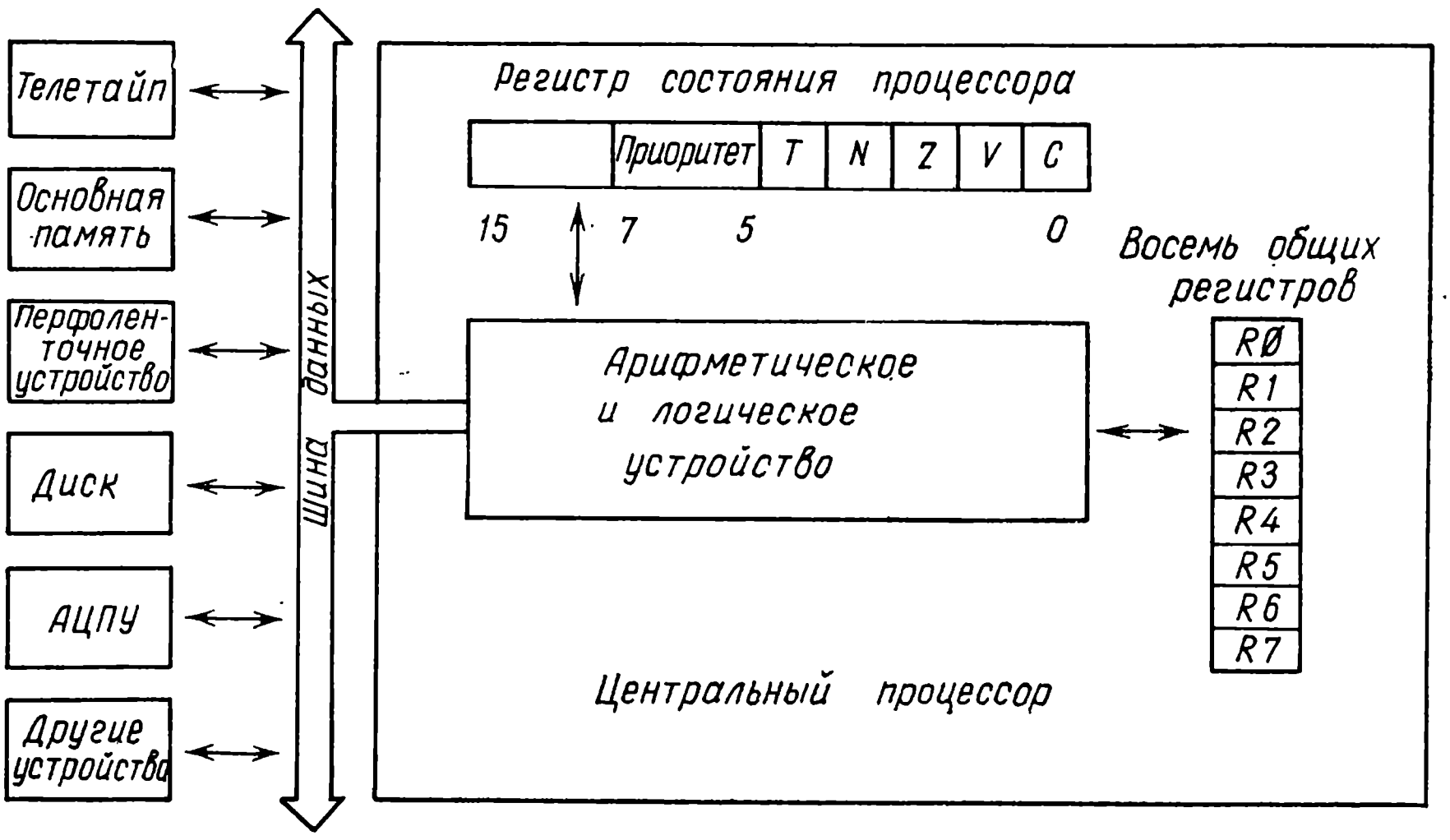


Рис. 3.1. Блок-схема системы PDP-11

Подобная гибкость позволяет программисту избирать для решаемой проблемы такие комбинации «команда/адрес», которые эффективны с точки зрения требуемой *памяти* и *времени* выполнения задачи. Это означает, что при любом алгоритме программист может создать относительно *компактный* (по сравнению с программами, реализующими этот же алгоритм на машинах с другой архитектурой) и быстрый (по сравнению с программами, выполняющимися на других машинах, одинаковых по технологии исполнения с соответствующими PDP-11) программный код.

PDP-11 являются параллельными ЭВМ, использующими для целых чисел (или чисел с фиксированной точкой) арифметику с дополнением до двух. PDP-11/60 включает также набор команд

и соответствующие регистры для манипуляции 32- или 64-битовыми числами с плавающей точкой. PDP-11/34 и 11/60 имеют *диспетчер* памяти, позволяющий обращаться с помощью 16-битового адреса к ячейкам памяти, расположенным за пределами 64 Кбайт. При необходимости этими устройствами может быть оснащено большинство других моделей семейства PDP-11.

Процессор PDP-11 включает прежде всего пять основных устройств, о которых уже говорилось в гл. 1. На рис. 3.1 показаны их составные части и взаимные связи (см. с. 55).

3.1. СТРУКТУРА МАШИНЫ PDP-11

Изображенные на рис. 2.1 упомянутые пять функциональных составляющих ЭВМ недостаточно ясно расшифровываются содержимым рис. 3.1, и поэтому представляется целесообразным более детально рассмотреть блоки ЭВМ PDP-11.

3.1.1. Концепция шины UNIBUS

На рис. 3.1 показаны типичные устройства ввода-вывода. Их связь с процессором (CPU) осуществляется посредством *шины данных,* представляющей собой набор проводов, передающих информационные сигналы (например, *адрес, данные* и *управляющие сигналы*) между подсоединенными к ней устройствами. Так как общая шина в состоянии одновременно передавать большое число сигналов (как, например, 16-битовое слово памяти), то на рис. 3.1 она показана широкой линией.

Шина подсоединена к общей магистрали потока данных компьютера. Любое устройство ввода или вывода, или CPU, может поместить информацию в магистраль, и, наоборот, каждому из них доступна информация из магистрали. Общая шина, таким образом, является эффективным средством передачи информации между функциональными устройствами, подсоединенными к ней.

Шина данных PDP-11, более известная как UNIBUS, позволяет всем устройствам ввода-вывода соединяться друг с другом для обмена информацией, равно как с процессором и основной памятью. Поскольку элементы системы связываются одинаково посредством UNIBUS, все устройства ввода и вывода становятся легко доступными одно для другого, в том числе для процессора и основной памяти. И так как UNIBUS способна обслуживать любое количество внешних устройств, то новые (например, дополнительный блок памяти или устройства ввода и вывода) могут быть просто подключены по мере их готовности.

Благодаря двунаправленной и асинхронной связи, обеспечиваемой UNIBUS, устройства могут посылать, получать данные и обмениваться ими независимо от центрального процессора. Например, дисплей с электронно-лучевой трубкой (CRT) может «обновлять» свой экран информацией из памяти, в то время как центральный процессор (CPU) выполняет другую задачу. Обладая

свойством асинхронности, UNIBUS совместима с устройствами, оперирующими в широком диапазоне скоростей.

Связь между устройствами по UNIBUS осуществляется с захватом шины. На каждую команду устройства-задатчика устройством-исполнителем после окончания передачи данных выдается сигнал ответа. Связь устройство — устройство не зависит ни от длины шины, ни от времени, необходимого на обмен ответными сигналами.

Многие устройства, подключенные к UNIBUS, работают и как вводные и как выводные. Диск, например, может использоваться для ввода через UNIBUS информации в машину.

В другом случае он же принимает выводную обработанную процессором информацию и размещает ее на хранение. Поэтому часто на такое устройство ссылаются как на один механизм ввода-вывода (I/O). В гл. 7 описываются методы передачи данных при вводе-выводе, но пока будем считать, что ЭВМ принимает информацию от устройств так, как это показано на блок-схеме, и возвращает ее обратно уже как выводную. Консоль PDP-11 позволяет программисту иметь прямой доступ к основной памяти и счетчику команд путем установки переключателей или на некоторых машинах — через посредство клавиатуры или терминальной консоли (см. приложение).

3.1.2. Арифметическое и логическое устройства

Арифметическое устройство — это центральная единица на блок-схеме PDP-11. Оно принимает данные из UNIBUS, из восьми общих регистров (*gp*) или из шести регистров плавающей точки (*fp*), если они имеются в архитектуре. Обработанную информацию это устройство может возвращать в те же источники данных и, кроме того, в регистр состояния процессора. Поскольку в PDP-11 для большинства арифметических операций используется двухадресный формат команд, нет необходимости иметь в ней регистр-аккумулятор. Вместо этого результат помещается в память, в общие регистры или в регистры плавающей точки.

3.1.3. Быстрая арифметика

Хотя операции расширенной арифметики для чисел с фиксированной точкой (например, умножение, деление, многократный сдвиг) могут быть запрограммированы с помощью основных логических команд и команд фиксированной точки, предоставляемых машиной PDP-11, этого недостаточно, если требуется высокоскоростная обработка данных. Поэтому к UNIBUS в качестве периферийного устройства может быть подключен *расширенный арифметический элемент* (EAE), значительно повышающий скорость выполнения операций умножения, деления, многократного сдвига и нормализации в арифметике с фиксированной точкой. Некоторые ЭВМ семейства PDP-11 имеют *расширенный набор команд*

(EIS), который также позволяет эффективно выполнять умножение, деление и многократный сдвиг в операциях арифметики с фиксированной точкой с помощью только общих регистров. При этом устраняется необходимость наличия EAE. Эффективная по времени манипуляция над числами с плавающей точкой осуществляется посредством *набора команд плавающей точки* (FIS), использующего в качестве аккумуляторов один или более стеков, или посредством *процессора плавающей точки* (EPP). Такой процессор имеет свое собственное множество команд и соответствующие аккумуляторы плавающей точки для быстрой 32- или 64-битовой арифметики плавающей точки.

EIS поставляется стандартно для PDP-11/34, 45, 50, 55, 60 и 70 и дополнительно для машин LSI-11 (11/03) и 11/40. Оборудование высокоскоростного процессора плавающей точки (FPP) является также дополнительно поставляемым для моделей 11/34, 45, 50, 55, 60 и 70. Будучи установленным на 11/60, оборудование FPP подменяет действия программного обеспечения при работе с плавающей точкой и на высокой скорости выполняет требуемые команды.

3.1.4. Устройство управления

Хотя устройство управления и не показано явно на рис. 3.1, его присутствие тем не менее обозначено регистром состояния, восемью общими регистрами и UNIBUS. Так, например, регистр 7 действует как счетчик команд PDP-11 (PC), в то время как биты N, Z, V и C регистра состояния характеризуют состояние машины, отражая результаты выполнения предыдущих команд.

Счетчик команд используется устройством управления PDP-11 для записи адреса ячейки памяти, содержащей команду, которая должна быть выполнена. Иными словами, PC всегда содержит адрес очередной подлежащей выполнению команды. Таким образом, при любом обращении процессора к счетчику команд для получения адреса слова памяти PC всегда увеличивается на 2, так как — это будет видно дальше — адрес слова всегда четный. Если встречается команда, требующая передачи управления на другую часть программы, то PC устанавливается на соответствующий адрес перехода. Изначально PC должен быть установлен на значение стартового адреса программы. Впоследствии он управляется командами программы.

Использование в PDP-11 общего регистра в качестве счетчика команд, вообще говоря, уникально. Обеспечиваемая при этом гибкость приводит к некоторым интересным способам спецификации относительного и непосредственного режимов адресации. Особенности применения счетчика команд для адресации операндов более детально обсуждаются позже в этой главе и гл. 6.

Регистр состояния процессора (PS) PDP-11 содержит информацию о результатах последней выполненной операции. Он разделен на однобитовые индикаторы, называемые *кодами условий* (CC). Индикаторы устанавливаются в нуль или единицу, в за-

висимости от окончательного значения величины, полученной при выполнении арифметической операции (например, TST, SUB, ADD и CMP) или манипуляции над словом (например, MOV). Индикаторы сохраняют состояние до тех пор, пока другая операция не изменит его. Анализ этих бит показывает, что Z-бит указывает на нулевой результат последней операции, а N-бит говорит о том, что он был отрицательным. Биты C и V определяются результатами, которые вызывают перенос и переполнение. Мы пока отложим их обсуждение, так же как и обсуждение T-бита и бит приоритета, до момента изучения гл. 4 и 7.

Необходимо обратить внимание на важность связи между кодами условий и множеством команд машины, поскольку первые играют важную роль при выполнении последних. Так, уменьшение счетчика повторения циклов в PDP-11 можно запрограммировать следующим образом:

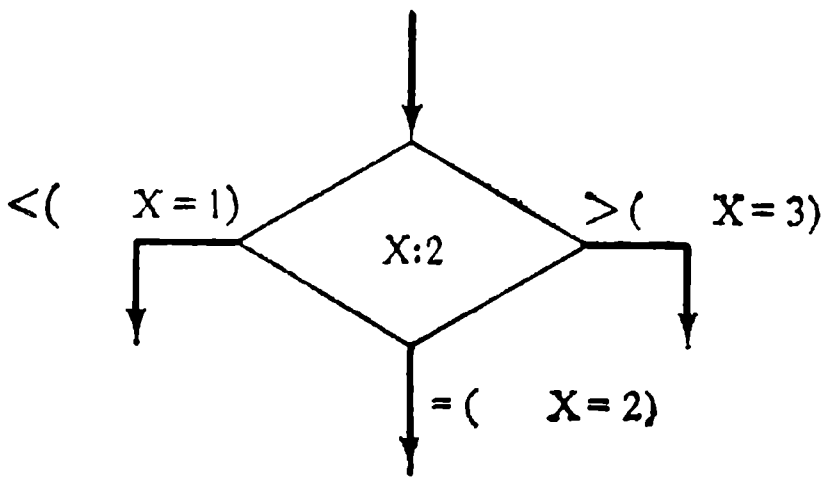
```
        MOV     #100,R0         ;NUMBER OF TIMES THROUGH LOOP
        MOV     #A,R1           ;ADDRESS OF A(1)
LOOP:   MOV     #0,(R1)+        ;CLEAR ARRAY LOCATION
        SUB     #1,R0           ;DECREMENT R0, SET Z BIT IF ZERO
        BNE     LOOP            ;LOOP UNTIL Z = 0
```

В отличие от примера в гл. 2 здесь отсутствует команда сравнения счетчика циклов с нулем. И так как подобная конструкция организации циклов встречается часто, большинство машин семейства PDP-11 имеет команду вычитания и перехода по условию, которая имеет следующую форму:

```
SOB     Rn,LOOP
```

где LOOP — символический адрес (адресующий некоторую точку программы, предшествующую расположению команды SOB), на который передается управление, если содержимое Rn, уменьшенное на единицу при выполнении SOB, *не равно* нулю.

Поскольку коды условий не изменяются командами перехода (или командой SOB), можно организовать разветвление процесса, применяя только команду сравнения. Так, если значение величины X находится между 1 и 3 включительно, то, базируясь на них, можно легко закодировать три ветви переходов:




```
        .
        .
        .
CMP     X,#2        ;TEST X
BMI     EQONE       ;X = 1
BEQ     EQTWO       ;X = 2
BPL     EQTHR       ;X = 3
        .
        .
        .
```

Не все команды PDP-11 воздействуют одинаково на биты условий регистра состояния процессора. В то время как команда CMP устанавливает в соответствии с условиями все биты (N, Z, V и C), команда MOV — только N и Z, а V она всегда устанавливает в нуль (т. е. безусловно). На бит C команда MOV не воздействует вообще. Эквивалентные по выполняемой операции команды, такие, как ADD #1,VALUE и INC VALUE (см. параграф 3.3.2, описывающий одноопе рандные команды), по-разному воздействуют на бит переноса C. Следовательно, программист должен быть особенно внимателен при работе как с множеством команд машин, так и с кодами условий, на которые они воздействуют.

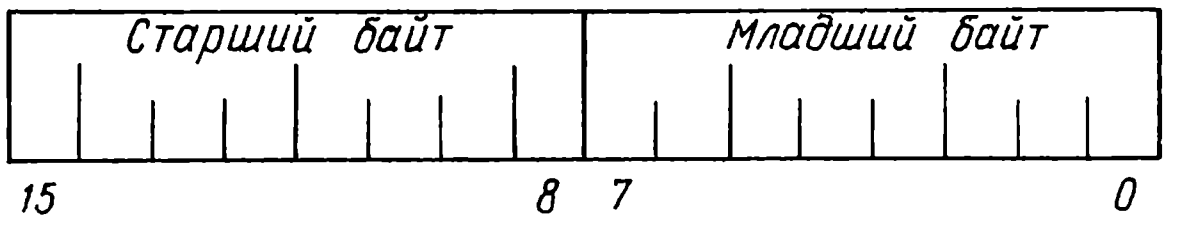
На базе кодов условий реализуются команды переходов по условиям. Есть две категории условных переходов — простая и знаковая, в которых тестируются индивидуальные биты условий и их комбинация для того, чтобы определить необходимость перехода. Команды перехода по условию BEQ, BNE, BMI и BPL, обсуждавшиеся в гл. 2, обеспечивают переход по простым условиям. Они тестируют одиночные биты (N или Z). Более сложные переходы — по знаку — будут обсуждаться ниже, в гл. 4, так как прежде надо рассмотреть понятия переноса и переполнения, без которых эти команды будут сложны для понимания.

3.1.5. Прямой доступ к памяти и страница ввода-вывода

Пространство адресуемой базовой памяти PDP-11 равно 65536 байтам (64 K, где K «означает» 1024), или 32 Kсловам. Верхние (со старшими адресами) 4 Kслова называются *страницей* или блоком *ввода-вывода* (I/O). В целом память можно представить как последовательность ячеек, каждой из которых присвоено число-адрес. Каждая ячейка в странице I/O является регистром управления или регистром данных, связанным с конкретным периферийным устройством. Как мы увидим, такая структура позволяет управлять устройствами без помощи специальных команд I/O.

Поскольку устройство памяти и команды PDP-11 позволяют обращаться как к 16-битовым словам, так и к 8-битовым половинам слов, называемым *байтами*, то общее количество адресов не соответствует числу слов. Каждое слово содержит 2 байта, и, следовательно, блок из 4096 слов содержит 8192 байта. Слова всегда имеют *четные адреса.* Возвращаясь к гипотетической машине гл. 2, мы теперь можем заметить одну важную разницу между ней и PDP-11. В PDP-11 адрес слова всегда *четная* величина. Поэтому следующие одна за другой команды всегда будут расположены по четным адресам и адресация слов-операндов производится *четной числовой величиной,* используемой как указатель адреса.

Слово в машине PDP-11 делится на старший и младший байты следующим образом:



Поскольку младшие байты хранятся в четных ячейках памяти, а старшие — в нечетных, легко увидеть, что адрес любого слова совпадает с адресом его младшего байта. Учитывая сказанное, память удобно представлять так, как это сделано на рис. 3.2.

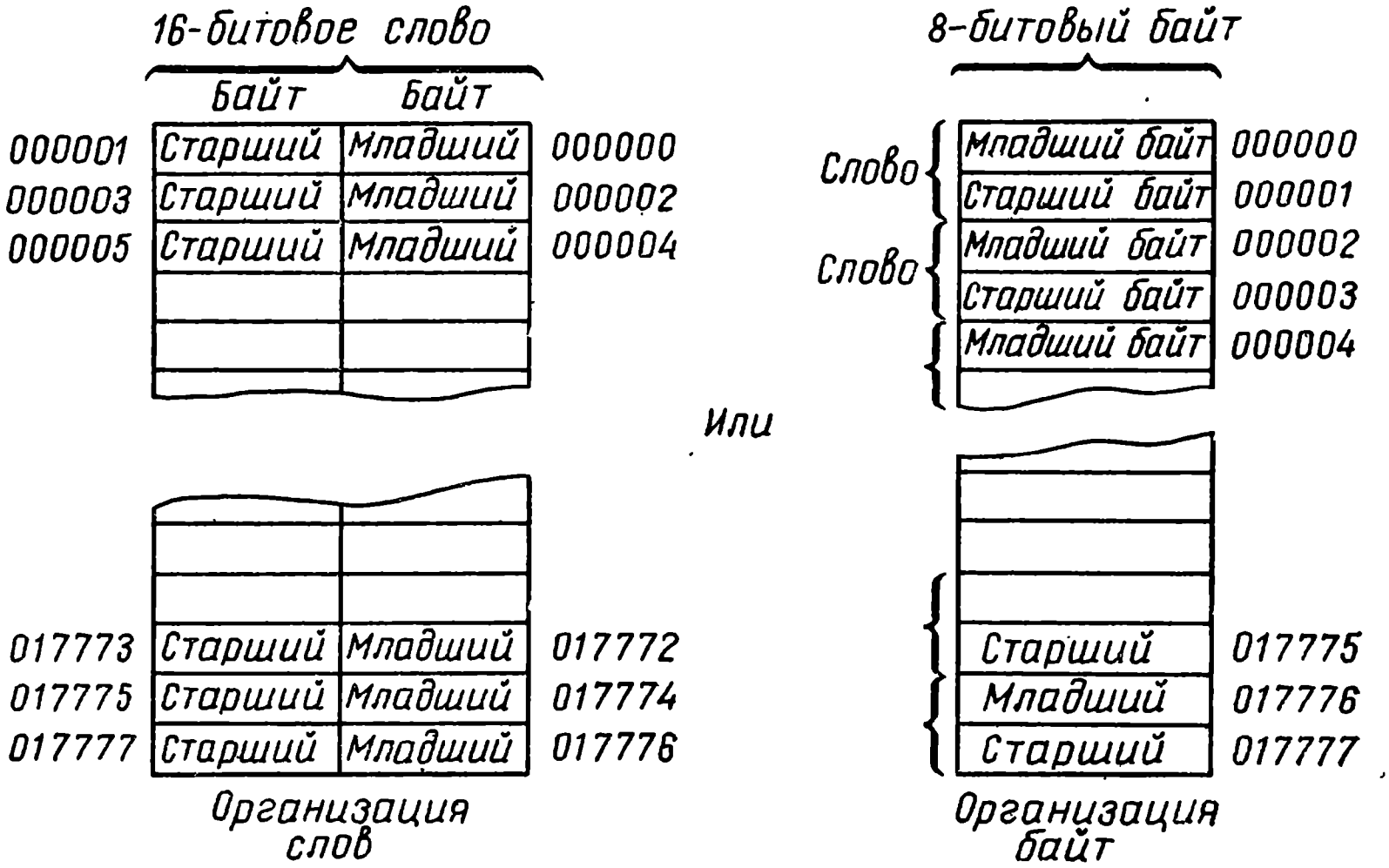


Рис. 3.2. Нумерация слов и байт в PDP-11

Определенные ячейки памяти резервируются для использования системой. Обычно они имеют адреса от 0 до 777(8) и, как уже отмечалось, верхние 4096 адресов слов (начиная с адреса 160000(8)). Применение этих ячеек описывается в гл. 7. Здесь и далее там, где это может вызвать двусмысленность, для указания основания системы счисления мы будем употреблять конструкцию «( )» или применять нижний индекс.

3.2. ПРЕДСТАВЛЕНИЕ ДАННЫХ

Данные в машине PDP-11 интерпретируются как принадлежащие к одному из следующих основных типов:

1) числа с фиксированной точкой (целые);
2) числа с плавающей точкой;
3) алфавитно-цифровые символы.

Прежде чем пытаться объяснять каждый тип, мы сделаем краткий обзор систем счисления.

3.2.1. Системы счисления

Важно понять, что данные, хранимые в памяти, неразличимы в том смысле, что все они являются просто двоичными числами. И только интерпретируя их, мы будем в состоянии использовать сответствующие информационные значения.

В *десятичной* системе счисления (основание 10) значение цифры зависит от ее позиции в числе, например:

$$\begin{array}{rcrcrcr} 347 & = & 3 & * & 100 & = & 300 \\ & + & 4 & * & 10 & = & 40 \\ & + & 7 & * & 1 & = & 7 \\ & & & & & & \overline{} \\ & & & & & & 347 \end{array}$$

Значение каждой позиции есть *позиционный коэффициент*. Каждый из них должен быть умножен на степень основания системы счисления, которая называется *весовым значением* или просто *весом*.

Принимая во внимание сказанное о позиционных коэффициентах и весах, можно выразить любую числовую систему, записываемую с весами, в следующей обобщенной форме:

$$X = x_n w_n + x_{n-1} w_{n-1} + \ldots + x_1 w_1 + x_0 w_0 + x_{-m} w_{-m},$$

где

$w_i = r^i$ (т. е. r^i — весовое значение и r — основание или база)

и

$0 \leqslant x_i \leqslant r - 1$ (т. е. x_i — позиционный коэффициент).

Очевидно, что наибольшее значение позиционного коэффициента всегда на единицу меньше значения основания системы счисления. Для основания (или базы) 2 это означает, что наибольший коэффициент есть 1, а для базы 10—9. Это может показаться и интуитивно очевидным, однако программист-новичок (а иногда и опытный) использует при кодировании программы недопустимые цифры (например, 10853 при основании 8 или 102 при основании 2).

Ниже записаны примеры полных формальных выражений для системы счисления с весом:

$$\begin{aligned} 132 &= 1*10^2 + 3*10^1 + 2*10^0 \\ &= 1*2^7 + 0*2^6 + 0*2^5 + 0*2^4 + 0*2^3 + 1*2^2 + 0*2^1 + 0*2^0 \\ &= 2*8^2 + 0*8^1 + 4*8^0 \\ &= 8*16^1 + 4*16^0 \end{aligned}$$

Другими словами,

$$132_{10} = 10000100_2 = 204_8 = 84_{16}$$

Интересно заметить, что, хотя все примеры подразумевали положительное основание системы счисления, оно может быть и отрицательным. Например, полагая, что база равна — 3, число 132 может быть выражено как

$$132_{10} = 2*(-3)^4 + 1*(-3)^3 + 0*(-3)^2 + 1*(-3)^1 + 0*(-3)^0$$

$$= 21010_{-3}$$

Можно придумать системы счисления без весов (и такие системы существуют, это «Excess-3» и «2 из 5»), но их обсуждение выходит за пределы данной книги.

3.2.2. Представление чисел в ЭВМ

Поскольку компонентам ЭВМ присуща внутренняя двоичная природа, все современные цифровые машины базируются на двоичной системе счисления. Не вдаваясь в обсуждение вопроса о том, насколько она целесообразна для ЭВМ, отметим ее чрезвычайную громоздкость при использовании людьми. Поэтому вместо двоичной системы большинство программистов пользуются арифметикой с базой 8 или 16, оставляя системным компонентам (ассемблерам, компиляторам, загрузчикам и др.) задачу преобразования таких чисел в их двоичный эквивалент. Современные ассемблеры позволяют вводить и десятичные числа, как целые, так и с плавающей точкой, что весьма удобно для большинства прикладных программ.

Преобразование двоичных чисел в *восьмеричные* (или числа с базой 8) и *шестнадцатеричные* (или с базой 16) достаточно легко выполняется. Для этого необходимо разбить двоичное число на группы по 3 бита (если преобразуем в восмеричный эквивалент) или по 4 бита (если преобразуем в шестнадцатеричный эквивалент), начиная с наименьшей значащей цифры (LSD), а затем заменить каждую двоичную группу соответствующим ей числом. Так, если имеется двоичное число 010011100001, то его восьмеричные и шестнадцатеричные эквиваленты равны:

```
010 011 100 001  = 2341
                 2       8
                 |
                LSD
и
  0100 1110 0001 = 4E1
                 2      16
```

Этот процесс настолько естествен, что большинство программистов может проделать требуемые преобразования в уме. Более того, для таких целей существуют специальные карманные калькуляторы, позволяющие работать как с восьмеричными, так и с шестнадцатеричными числами. Их удобно использовать при отладке программ.

3.2.3. Отрицательные числа

Есть три пути для представления отрицательных чисел с любым основанием. Например, двоичные отрицательные числа могут быть представлены в трех формах: *знаковой, с дополнением до единицы* и *с дополнением до двух.* Какую же форму использует ЭВМ при выполнении арифметических операций?

Обычно предпочтение отдается формам с дополнением. Это обусловлено тем, что при использовании знаковой формы очень сложной становится реализация операций сложения и вычитания. Вопрос о том, какую из двух оставшихся форм выбрать, остается открытым, и все зависит от решений, принимаемых дизайнерами ЭВМ и системными программистами.

Генерация чисел в форме «с дополнением до единицы» осуществляется легче, чем в форме «с дополнением до двух». К тому же с точки зрения оборудования компьютера арифметическое устройство в этом случае имеет более «стандартный» вид, чем при использовании арифметики «с дополнением до двух». С другой стороны, запись чисел в форме «с дополнением до единицы» допускает два представления нуля (отрицательный нуль и положительный нуль):

```
000 000    нуль
111 111    минус нуль в дополнении до единицы
```

в то время как в форме представления «с дополнением до двух» существует единственный нуль:

```
            000 000    нуль
            111 111    дополнение до единицы
                +1     плюс единица
            ---------
пропадает 1) 000 000   нуль в дополнении до двух
```

С точки зрения математики не очень удобно иметь два представления нуля, поэтому в большинстве современных машин для представления отрицательных чисел применяют форму «с дополнением до двух».

3.2.4. Числа с фиксированной точкой

Целые числа в PDP-11 представляются *байтами, словами* или *двойными словами.* Их размеры, гипы и диапазоны показаны ниже:

Тип	*Длина*	*С учетом знака* ←*ДИАПАЗОН*→	*Без учета знака*
Byte	8 bits	-128 to +127	0 to 255
Word	16 bits	-32768 to +32767	0 to 65535
Longword	32 bits	-2**31 to +2**31-1	0 to 2**32-1

Как мы увидим, PDP-11 имеет полный набор команд сложения, вычитания, умножения, деления, дополнения и сдвига *слов.* Все они трактуют слово как число с дополнением до двух (со знаком). Аналогично над *байтами* с помощью команд для слов можно производить операции сложения, вычитания, умножения или деления.

Однако в некоторых арифметических операциях байтами можно манипулировать прямо, применяя байтовые инструкции. В двойные (*длинные*) слова помещается произведение (при умножении) или делимое — при выполнении операции деления. Над двойными словами допустимы операции прямого арифметического сдвига, сложения или вычитания слова посредством команд сложения/вычитания слов. Набор команд, обеспечивающий условный переход по знаку, позволяет в программе передавать управление в зависимости от результатов арифметических операций. Заметим, что самый старший бит в числе с дополнением до двух всегда указывает знак числа. Тестируя его, можно организовывать требуемую передачу управления.

Среди команд машины PDP-11 имеются и такие, которые трактуют целые числа как величины без знака. Они полезны при манипуляции адресами.

3.2.5. Числа с плавающей точкой

Целые числа полезны для представления данных и решения задач во многих областях. Однако часто они не обладают *динамическим диапазоном,* необходимым для получения правильных решений в сфере научных приложений. Например, хотя целые числа могут представлять данные с достаточной *точностью*, нередко *величина* числа или промежуточный результат могут превышать диапазон, определяемый длиной слова. Число с *плавающей точкой* подобно целому образуется последовательностью смежных бит в памяти. Однако они интерпретируются как имеющие две части: *мантиссу* и *порядок.* Таким образом, число с плавающей точкой может храниться в слове компьютера в следующем виде:

s	порядок	s	мантисса

и интерпретироваться как $\pm.nnnnnnnn * 2 ** (\pm mmmmmmmm)$, где «*n*» — биты мантиссы и «*m*» — порядка. Заметим, что мы произвольно разместили два знаковых бита так, чтобы обозначить положительную и отрицательную мантиссу и порядок. Однако обычно в ЭВМ представление чисел с плавающей точкой в явном виде включает только один знак. При этом старший значащий бит (MSB) считается знаком мантиссы и делит диапазон возможных значений порядка на две половины: одну — для отрицательных, другую — для положительных значений. Мы продемонстрируем этот способ на следующем примере.

Пусть величина 46,5 должна быть представлена как величина с плавающей точкой в 16-битовом машинном слове. Биты в слове имеют представленное ниже назначение:

1	5 бит	10 бит
s	порядок	мантисса

↑ — двоичная точка

Переводя число из одной системы счисления в другую, получим, что целая и дробная части десятичного числа 46,5 в восьмеричном виде примут значение 56,4. Представляя его как двоичное, получим в результате число с плавающей точкой:

```
46.5    = 56.4    =    .564*(8**2) = .101110100*(2**6)
```

Для того чтобы выразить этот двоичный результат в формате числа с плавающей точкой, необходимо разделить диапазон значений порядка на две части:

```
1111
 •      положительные порядки
1000    порядок равен нулю
 •      отрицательные порядки
0000
```

Таким образом, число 2 ** 4, т. е. 16, интерпретируется как значение порядка, равное нулю, т. е. здесь применяется система счисления «*excess-16*».

Число 46,5, мантисса которого равняется 101110100, а порядок равен 10000 + 110 = 10110, может быть представлено как

```
0 10110 1011101000
```

Фактические значения чисел с плавающей точкой могут находиться в диапазоне 0,1777 (8) * 2 ** 15 (около 32000) для наибольшей положительной величины и 0,1000(8) * 2 ** (—16) (около 0,00001) — для наименьшей положительной величины. Отрицательные числа находятся в этом же диапазоне. Хотя общие пределы представления целых чисел и чисел с плавающей точкой одинаковы, различие состоит в том, что в формате с плавающей точкой можно выражать дробные числа. Однако применение такого формата для выражения числа использует только 10 бит (в отличие от 16 для формата представления целого числа), что уменьшает точность числа с плавающей точкой при одной и той же длине слова.

Известно, что при манипуляциях числами с плавающей точкой обычно необходимо выравнивание порядков до выполнения таких операций, как сложение или вычитание:

```
15.75*10**2 = 157.5*10
42.5 *10**1 =  42.5*10
                --------
               200.0*10 = 20.0*10**2
```

В ЭВМ также необходимо масштабирование порядков, чтобы при выполнении действий сложения или вычитания чисел с плавающей точкой можно было эффективно применять арифметику чисел с фиксированной точкой.

Однако, скажем, когда вычитаются два числа, близкие по значению, возникают серьезные проблемы. Например:

```
 46.75 = 0 10110 1011101100
-46.50 = 0 10110 1011101000
-----    -----------------
   .25 = 0 10110 0000000100
```

В этом случае говорят, что результат *не нормилизован.* Число считается *нормализованным,* если первый бит его дробной части является значащим. В примере, приведенном выше, нормализованная форма получается путем умножения дробной части на соответствующую степень двух и вычитания этой же степени двух из экспоненты:

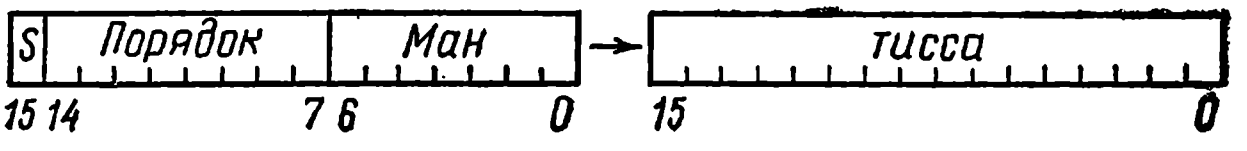
```
.25 = 0 01111 1000000000 = 2**(-1)*0.5
```

Нормализация применяется по двум причинам. Во-первых, она позволяет использовать максимальное количество бит для представления дробной части и, во-вторых, упрощает процесс выполнения операций над числами с плавающей точкой.

Существуют два исключения в форматах нормализованного числа. Первое: значение нуль хранится как общий повсеместный нуль. Второе: степени двух хранятся со значением знакового бита, равным 1, а дробной части — 10000... (т. е. MSB = 1). Таким образом, величины —(2 ** —1), —(2 ** 2), —(2 ** 0) и —(2 ** —3) в случае представления их в 16-битовой форме числа с плавающей точкой будут храниться в следующем виде:

```
-1/2 = 1 10000 1000000000
-4   = 1 10011 1000000000
-1   = 1 10001 1000000000
-1/8 = 1 01110 1000000000
```

В PDP-11 числа с плавающей точкой одинарной точности хранятся в *двух* прилегающих друг к другу 16-битовых словах, что позволяет увеличить точность представления дробной части и динамический диапазон числа по сравнению с 16-битовым представлением целых чисел. Бит 15 старшего слова предназначен для знака числа, следующие 8 бит обозначают *порядок* с «excess-128» и отделяют старшую часть мантиссы от ее знака. Младшая часть мантиссы хранится в следующем, старшем по адресу, 16-битовом слове:



Попытаемся формально выразить все сказанное выше для 32-битовых чисел с плавающей точкой одинарной точности, хранимых в PDP-11:

1. Мантисса *fr* представляется в виде нормализованного 24-битового положительного значения $0{,}5 \leqslant fr < 1$ с двоичной точкой, расположенной слева от ее первого значащего бита. *Поскольку этот бит должен быть равен 1, он не входит в число.* Это позволяет хранить *fr* в 23 битах.

2. Порядок *e* задается как смещенное 8-битовое положительное целое. Это означает, что когда из *e* вычитается 128, то результат представляет степень 2, на которую умножается мантисса, чтобы получилось истинное значение числа с плавающей точкой.

3. Знак числа считается положительным, когда знаковый бит $s = 0$, и отрицательным, когда $s = 1$. Таким образом, число, представленное 32-битовым словом, будет иметь вид

$$X = (1-2*s)*fr*2**(e-128),$$

где $2**(-128) = 2{,}939*10**(-39) \leqslant |X| \leqslant 2**(127) = 1{,}701*10**38$. Биты, отведенные для *fr*, гарантируют вычисления с точностью до седьмого знака для десятичных цифр; кроме того, в случае использования чисел с плавающей точкой двойной точности к данному слову добавляется еще одно 32-битовое слово, содержащее около 16 десятичных цифр дробной части. Заметим, что если $e = 0$, то число X считается равным 0.0, несмотря на то, что *s* или *fr* не равны нулю.

Применение FIS и FPP для манипуляции числами с плавающей точкой будет анализироваться в разд. 5.4.

3.2.6. Алфавитно-цифровые символы

Алфавитно-цифровые символы хранятся в ЭВМ в виде *байт,* содержащих уникальное цифровое представление каждого символа. В PDP-11 применяется код ASCII. В нем числа 0—9 кодируются в восьмеричном коде как 60—71, символы верхнего регистра — как 101—132, а буквы нижнего регистра — как 140—172. В слове хранятся два символа кода ASCII. Так, например, два символа AB хранятся как

Старший байт Младший байт

Значение B = 1 0 2 | 1 0 1 = Значение A

0 1/00 0/01 0/ 01/ 000/ 001

0 4 1 1 0 1

Это означает, что «AB» = 041101.

3.3. ФОРМАТЫ КОМАНД И РЕЖИМЫ АДРЕСАЦИИ PDP-11

Доступ к данным, хранимым в памяти, а также манипуляция ими в PDP-11 обеспечиваются командами, обычно специфицирующими:

1) функцию, которую необходимо исполнить (т. е. код операции);

2) общий регистр, используемый для определения операнда-источника и/или операнда-приемника;

3) режим адресации (для указания того, как избранный регистр должен применяться).

Программирующему на PDP-11 особо важно знать форматы команд и способы адресации данной машины. Каждое поле команды играет важную роль в том смысле, что оно определяет различные внешние характеристики машины (например, число регистров, объем прямо адресуемой памяти, количество кодов операций). В отличие от машин со старой архитектурой, где программист адресовал прямо только небольшие участки памяти длиной 256 (8-битовое адресное поле) или 4096 (12-битовое адресное поле) слов, PDP-11 позволяет программисту применять полный 16-битовый адрес. При этом может использоваться непосредственная, прямая, косвенная, индексная или автоиндексная адресация с применением восьми общих регистров. Таким образом, PDP-11 имеет весьма общую (вначале кажущуюся избыточной) структуру адресации, которая кроме перечисленных включает и *относительную адресацию* (т. е. адреса определяются по отношению к текущему значению счетчика команд), которая может быть реализована при написании программ, не зависящих от места расположения в памяти. (Эта тема будет обсуждаться в гл. 4.)

Так как большая часть данных, которыми манипулирует ЭВМ, обычно структурирована (символьные строки, массивы, списки, и т. д.), ЭВМ должна разрабатываться так, чтобы оперировать ими эффективно и гибко. Программист использует общие регистры в качестве следующих элементов вычислений:

1) аккумуляторов, в которых находятся данные для манипуляции;

2) указателей данных (в противоположность функции аккумуляторов);

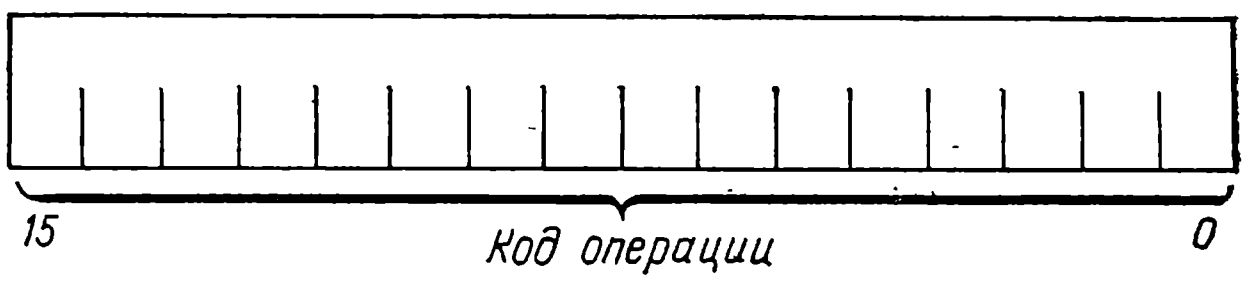
3) индексных регистров, чтобы указывать относительное положение элементов данных в таблице или списке.

Применение регистров в PDP-11 как для манипуляции данными, так и для вычисления адресов приводит к форматам команд переменной длины. Например, если для спецификации источника и/или приемника используются только регистры, то требуется одно слово памяти для хранения команды (в слове хранятся код операции и спецификатор операнда-источника и/или операнда-приемника). Однако, если употреблять их как указатели памяти, для индексации или в качестве составных частей команд, адресующихся к памяти, то требуются два или более слова для хранения таких команд (для кода операции, режима адресации, задания регистров и полного спецификатора адреса). Дополнительные слова, следующие в памяти за командой, служат для адресации всех 32 768 слов (т. е. 2**15 слов) или 65 536 байт (т. е. 2**16 байт) памяти.

Так как команды могут быть длиной в одно, два или три слова, PDP-11 иногда описывают как *процессор с переменной длиной команд.* Их формат требует от программиста указания определенного режима адресации. PDP-11 имеет команды четырех основных форматов: управляющие, однооперандные, двухоперандные и команды переходов.

3.3.1. Команды управления

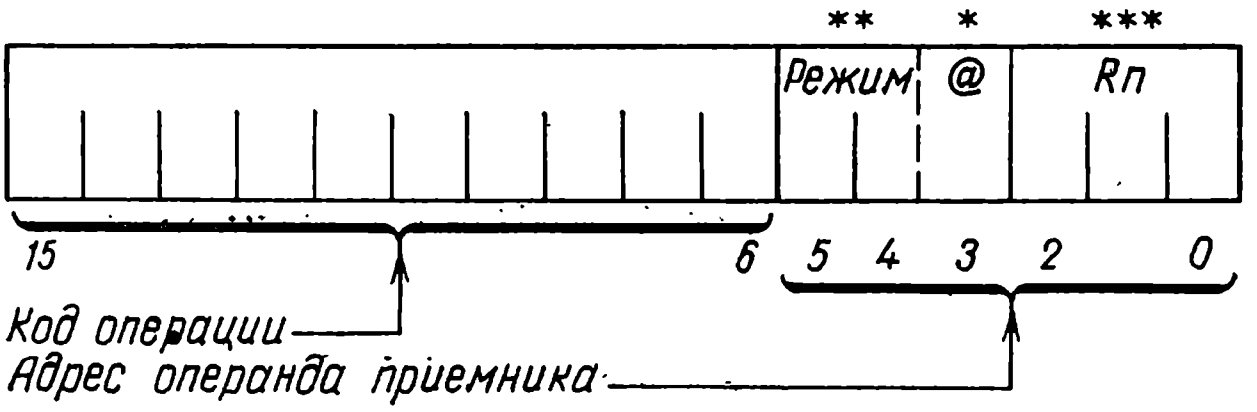
Команды управления имеют следующий формат:



Здесь все биты определяют код операции, имеющий длину, равную одному слову. Первой в этой группе является команда HALT с восьмеричным кодом операции 000000. Далее идут WAIT, RESET, IOT, RTI и EMT. Анализ их применения будет дан позже.

3.3.2. Однооперандные команды

Формат первого слова всех однооперандных команд имеет следующий вид:



* Специфицирует прямой или косвенный адрес
** Специфицирует, как будет использоваться регистр
*** Специфицирует один из 8 общих регистров

Биты от 15-го до 6-го специфицируют код операции, т. е. тип команды, которая должна быть выполнена. Биты от 5-го по 0-й формируют 6-битовое поле, называемое *полем адреса приемника.* Оно состоит из двух подполей:

1. *Подполе спецификации регистра.* Биты 0—2 специфицируют, какой из восьми общих регистров будет использоваться этой командой.

2. *Подполе спецификации адресного режима.* Биты 4—5 специфицируют, как будет применяться избранный регистр. Бит 3 указывает на прямой или косвенный режим адресации.

Таким образом, адрес назначения может специфицироваться одним из восьми способов, с помощью одного из восьми общих регистров. Примечательно, что 16-битовый адрес эффективно

определяется с помощью регистра и указания, как интерпретировать его содержимое. В некоторых случаях для определения адреса служит 16-битовое слово памяти вместе с содержимым регистра. Из этого следует, что в зависимости от специфицированного режима адресации такие команды могут быть длиной как в *одно*, так и в *два* слова. Второе слово генерируется ассемблером и содержит 16-битовый адрес или константу, которая применяется для формирования действительного адреса указанного операнда. В табл. 3.1 приведено подмножество одноопернадных команд, имеющихся в PDP-11.

Таблица 3.1. Одноопернадные команды

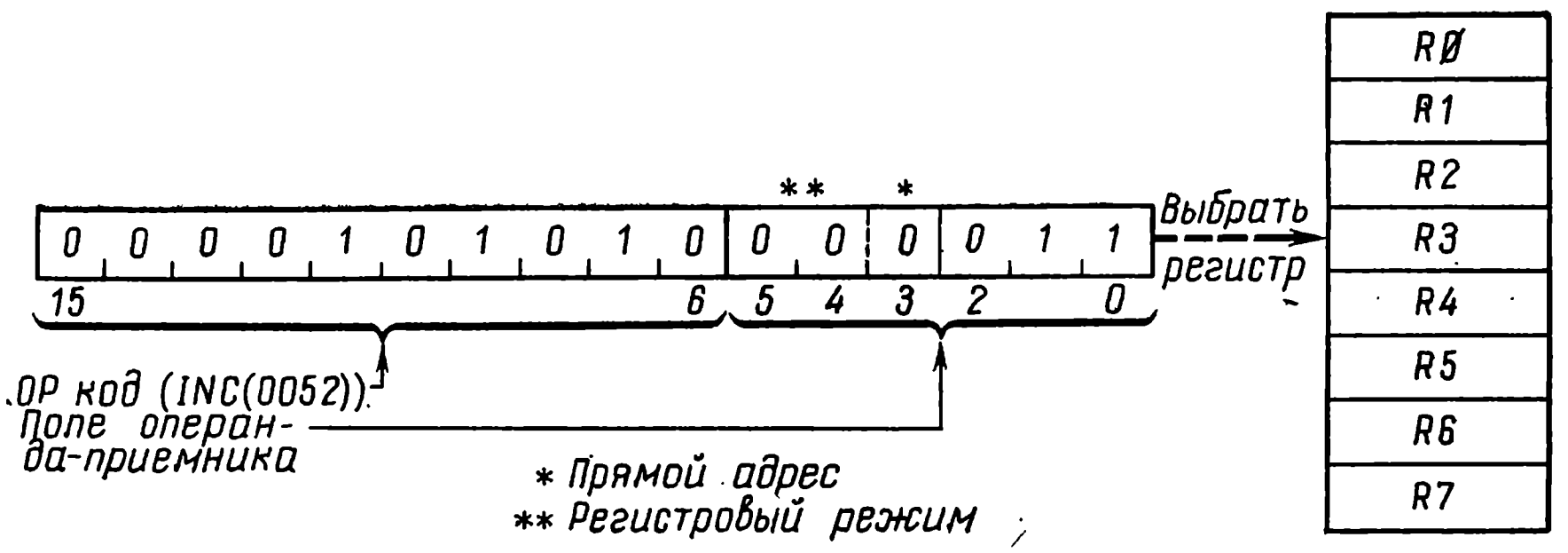
Операции	*Символическая форма*	*Действие*
Clear	CLR	(DST) ← 0
Complement	COM	(DST) ← ~(DST) i.e., logical complement
Increment	INC	(DST) ← (DST) + 1
Decrement	DEC	(DST) ← (DST) - 1
Negat	NEG	(DST) ← -(DST) i.e., arithmetic complement
Test	TST	Check (DST), Set Condition Codes
Jump	JMP	(PC) ← DST
Swap Bytes	SWAB	(DST)[15:8] ← (DST)[7:0] and, simultaneously, (DST)[7:0] ← (DST)[15:8]

Обозначение $[m:n]$ указывает на то, что биты от m до n включительно, входящие в операнд, подлежат манипуляции.

В следующем разделе для иллюстрации всех режимов адресации используется большое число команд. Поэтому представляется целесообразным ввести описания употребляемых обозначений. Как и ранее, «( )» означает *содержимое* регистра или ячейки памяти. Все его изменения, вызванные действием команды, будут *отмечены* особо. Предполагается, что команды, приводимые в примерах, занимают последовательные ячейки памяти, начиная с адреса 1000(8). Большинство из этих команд имеют модификации для операций с байтами.

Прямая адресация (режимы 0, 2, 4, 6): примеры одноопернадных команд

Регистровая адресация (режим 0). Однословная команда «INC R3» имеет восьмеричный код 005203 и двоичное представление.



Регистр содержит операнд; такая адресация называется *регистровой*. Так, если в этом примере R3 содержит число 5, то до и после выполнения команды имеем:

До

```
address space   register
(001000)=005203 (R3)=000005
```

После

```
address space   register
(001000)=005203 (R3)=000006
```

Автоинкрементная/автодекрементная адресация (режимы 2 и 4). Регистр содержит адрес операнда (т. е. служит указателем). Содержимое регистра может быть автоматически увеличено (*автоинкрементный режим*) или уменьшено (*автодекрементный режим*). В обоих случаях оно изменяется автоматически, указывая на следующую (с большим или меньшим адресом) ячейку. В табл. 3.2 показаны примеры однооперандных однословных команд. Читатель заметит, что автоуменьшение происходит *прежде*, чем регистр используется как указатель адреса, а автоувеличение — *после*. Версии этих команд для работы с байтами в мнемонике кода операции имеют суффикс «B» и увеличивают или

Таблица 3.2. **Автоинкремент (режим 2)/автодекремент (режим 4)**

Символьный вид	Восьмеричный код команды	Описание
Автоинкремент NEG (R5)+	005425	Содержимое R5 используется как адрес операнда. Значение операнда заменяется на его дополнение до 2. Затем содержимое R5 увеличивается до 2.
ДО память (001000) = 005425 . . (030000) = 000001	регистр (R5) = 030000	ПОСЛЕ память (001000) = 005425 . . (030000) = 177777 регистр (R5) = 030002
Автодекремент COM −(R3)	005143	Содержимое R3 уменьшается на 2 и используется как адрес операнда. Операнд заменяется его логическим дополнением или дополнением до единицы.
ДО память 001000 = 005143 . . (001774) = 000000	регистр (R3) = 001776	ПОСЛЕ память (001000) = 005143 . . (001774) = 177777 регистр (R3) = 001774

уменьшают на единицу, а не на два содержимое соответствующего регистра.

Индексная адресация (режим 6). Последний пример прямого использования регистра демонстрирует индексирование. В *индексном режиме* содержимое регистра и инлексное слово, следующее за командой, суммируются для получения адреса операнда. И поскольку в данном случае требуется 16-битовое индексное слово, то, следовательно, необходимы два слова для размещения этой однооперандной команды (см. табл. 3.3 и 3.4).

Таблица 3.3. **Индексный режим (режим 6)**

Символьный вид	Восьмеричный код команды	Описание
CLR 200 (R4)	005064 000200	Адрес операнда определяется путем добавления 200 к содержимому регистра R4. Ячейка памяти по этому адресу очищается.
ДО память (001000) = 005064 (001002) = 000200 . . (002200) = 123456	регистр (R4) = 002000	ПОСЛЕ память (001000) = 005064 (001002) = 000200 . . (002200) = 000000	регистр (R4) = 002000

Таблица 3.4. **Суммарная таблица основных режимов прямой адресации**

Двоичный код	Наименование режима	Обозначение на Ассемблере	Функция
000	Регистровый	Rn	Регистр содержит операнд.
010	Автоинкрементный	(Rn)+	Регистр содержит адрес операнда, затем увеличивается.
100	Автодекрементный	—(Rn)	Регистр уменьшается, затем его содержимое берется как адрес операнда.
110	Индексный	X (Rn)	Величину X (находящуюся в ячейке непосредственно за командой) добавляют к содержимому Rn, чтобы получить адрес операнда.

Косвенная адресация (режимы 1, 3, 5 и 7)

Если бит 3 команды установлен, то, значит, специфицирована косвенная адресация, включающая четыре основных режима. *Режим* называется *регистровым косвенным*, если содержимое избран-

ного регистра рассматривается как адрес операнда. В случае *автоинкрементного/автодекрементного косвенного* режима оно используется (перед увеличением или после уменьшения) как адрес адреса операнда. (Заметим, что в PDP-11 адрес памяти всегда указывается 16-битовым словом. Поэтому содержимое регистра *всегда* увеличивается или уменьшается на два вне зависимости от того, на какую границу — слова или байта — оно указывает.) И наконец, при *индексном косвенном режиме* мы получаем адрес адреса операнда, вычисленный как сумма содержимого регистра и слова, следующего за командой.

Таблица 3.5. **Регистровый косвенный режим (режим 1)**

Символьный вид	Восьмеричный код команды	Описание
INC @R5 или INC (R5)	005215	Увеличивается содержимое ячейки, на которую указывает R5
ДО память (001000) = 005215 . . . (001700) = 000100	 регистр (R5) = 001700	ПОСЛЕ память (001000) = 005215 . . . (001700) = 000101	 регистр (R5) = 001700

Таблица 3.6. **Автоинкрементный косвенный режим (режим 3)**

Символьный вид	Восьмеричный код команды	Описание
NEG @(R2)+	005432	Содержимое R2 используется как адрес операнда, который заменяется на дополнение до двух. Затем содержимое R2 увеличивается на 2.
ДО память (001000) = 005432 . . (001010) = 000001 . . (010300) = 001010	 регистр (R2) = 010300	ПОСЛЕ память (001000) = 005432 . . (001010) = 177777 . . (010300) = 001010	 регистр (R2) = 010302

При написании программ на языке Ассемблера на косвенную адресацию указывает знак «@» (или скобки, если это не приводит к двусмысленности), который ставится, как префикс, перед спецификацией операнда. Табл. 3.5, 3.6, 3.7 и 3.8 иллюстрируют применение всех четырех косвенных режимов адресации.

Таблица 3.7. **Автодекрементный косвенный режим (режим 5)**

Символьный вид	Восьмеричный код команды	Описание
COM @ — (R0)	005150	Содержимое R0 уменьшается на 2, затем используется как адрес адреса операнда. Значение операнда заменяется на его дополнение до единицы.
ДО память (001000) = 005150 . . (010100) = 000000 . . (010774) = 010100	регистр (R0) = 010776	ПОСЛЕ память (001000) = 005150 . . (010100) = 177777 . . (010774) = 010100 регистр (R0) = 010774

Таблица 3.8. **Индексный косвенный режим (режим 7)**

Символьный вид	Восьмеричный код инструкции	Описание
CLR @ 1000(R2)	005072 001000	Содержимое R2 и индексное слово суммируются и образуют адрес адреса операнда. Операнд очищается.
ДО память (001000) = 005072 (001002) = 001000 . . (001050) = 007777 . . (001100) = 001050	регистр (R2) = 000100	ПОСЛЕ память (001000) = 005072 (001002) = 001000 . . (001050) = 000000 . . (001100) = 001050 регистр (R2) = 000100

Табл. 3.9 суммирует эти же четыре режима косвенной адресации.

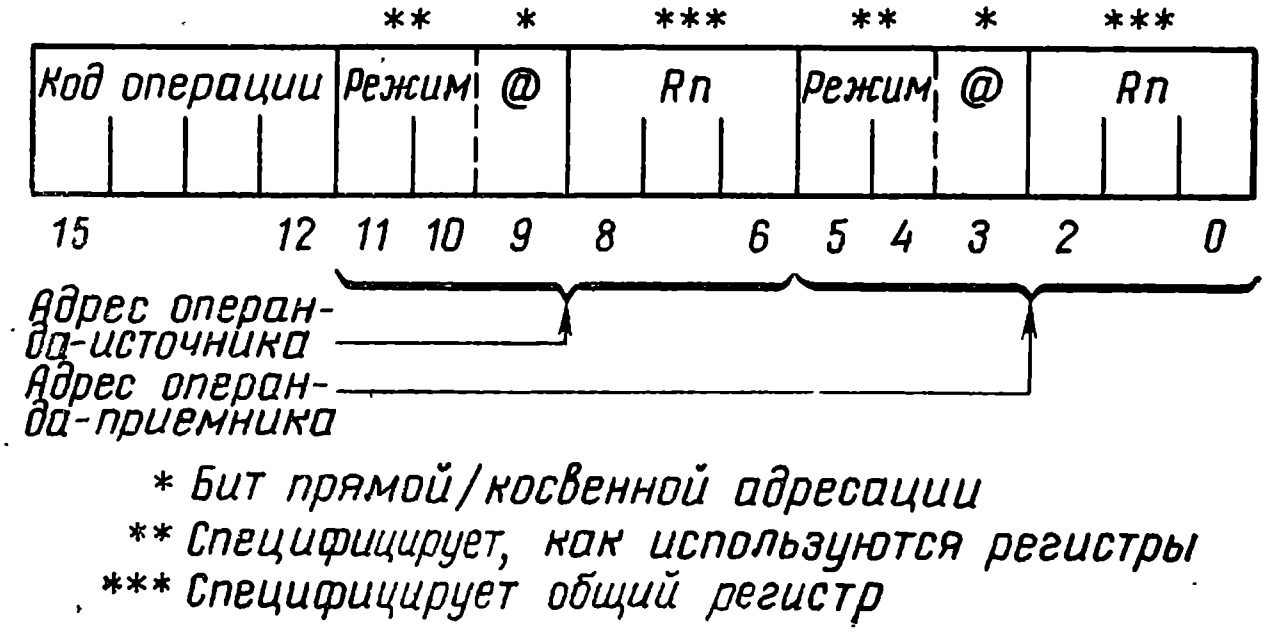
Таблица 3.9. **Косвенные режимы**

Двоичный код	Наименование режима	Обозначение на Ассемблере	Функция
001	Регистровый косвенный	@Rn или (Rn)	Регистр содержит адрес операнда.
011	Автоинкрементный косвенный	@(Rn) +	Регистр указывает на слово, которое содержит адрес операнда. Затем регистр увеличивается на 2, даже если команда работает с байтами.
101	Автодекрементный косвенный	@—(Rn)	Регистр уменьшается на 2, даже в случае, если команда работает с байтами. Затем его содержимое используется как указатель слова, содержащего адрес операнда.
111	Индексный косвенный	@X (Rn)	Величина X (хранящаяся в слове, непосредственно следующем за командой) суммируется с содержимым Rn. Полученное значение используется как адрес слова, содержащего адрес операнда. Ни X, ни Rn не модифицируется.

Итак, для любого вида адресации, прямого или косвенного, биты режима определяют фактическую длину команды (например, одно или два слова). И только когда используется индексный режим, команда всегда занимает два слова.

3.3.3. Двухоперандные команды

Операции, которые предполагают два операнда (такие, как сложение, вычитание, пересылка и сравнение), реализуются командами, требующими двух адресов. Первый операнд называется *операндом-источником*, второй — *операндом-приемником*. Задавая различные назначения бит в поле операнда-источника и операнда-приемника, мы можем специфицировать различные регистры и различные режимы адресации. Двухоперандная команда имеет следующий формат:



Здесь поле адреса источника предназначено для определения источника или первого операнда; адрес приемника указывает размещение второго операнда и результата. Так, например, команда ADD A,B прибавляет содержимое (операнд-источник) ячейки A к содержимому (операнд-приемник) ячейки B. После ее выполнения ячейка B содержит сумму, содержимое ячейки A остается без изменения. Подмножество двухоперандных команд PDP-11 приведено в табл. 3.10.

Таблица 3.10. **Двухоперандные команды**

Операция	Символическая форма	Действие
Move	MOV	(DST) ← (SRC).
Add	ADD	(DST) ← (DST)+ (SRC).
Sub	SUB	(DST) ← (DST) — (SRC).
Compare	CMP	Вычисляет (SRC) — (DST), затем модифицирует коды условий.
Bit set	BIS	(DST) ← (SRC) ∨ (DST), включающее OR операндов.
Bit clear	BIC	(DST) ← ~ (SRC) ∧ (DST). Очищает каждый бит в поле операнда приемника, если установлен соответствующий бит операнда-источника.
Bit test	BIT	Выполняет логическое AND операндов, модифицирует коды условий.

Пример двухоперандных команд

Так как для группы двухоперандных команд значение бит режима то же, что и для однооперандных, они являются также

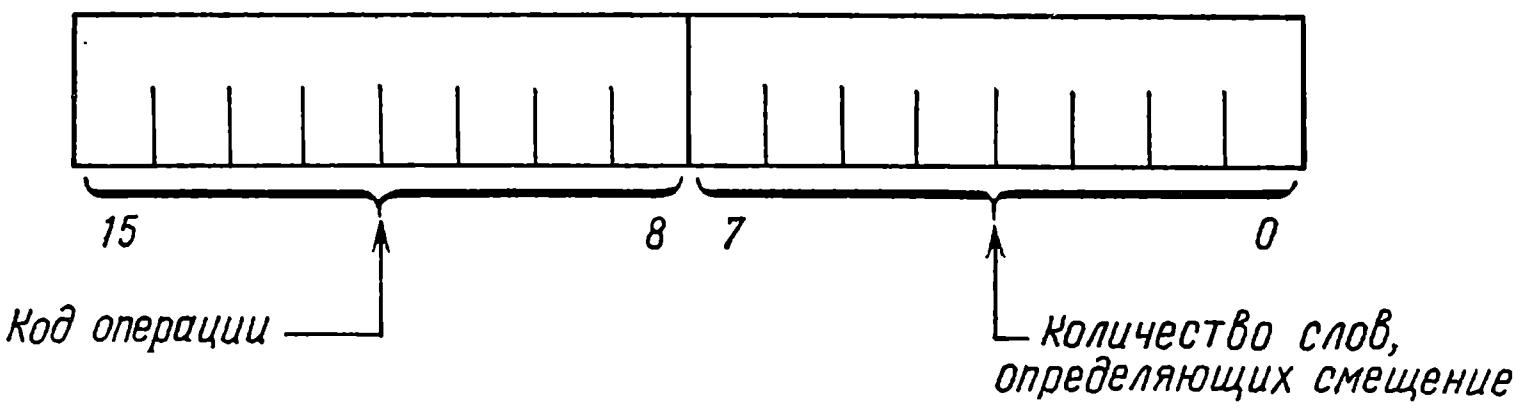
Таблица 3.11. **Пример двухоперандной команды**

Символьный вид	Восьмеричный код команды	Описание
ADD 30 (R2), (R1)+	066221 000030	Содержимое R2 и индексного слова суммируется, определяя адрес операнда-источника. Содержимое по этому адресу добавляется к операнду, адрес которого указан в R1. R1 затем увеличивается на 2.
ДО память (001000) = 066221 (001002) = 000030 ⋮ (001130) = 000001 ⋮ (001500) = 000025	регистр (R1) = 001500 (R2) = 001100	ПОСЛЕ память (001000) = 066221 (001002) = 000030 ⋮ (001130) = 000001 ⋮ (001500) = 000026
		регистр (R1) = 001502 (R2) = 001100

командами переменной длины и могут быть одно-, двух- или трехсловными. И вычисление *действительных адресов* операндов производится точно так же, как и для случая одноопе рандных команд. В табл. 3.11, показан пример двухсловной команды ADD, в которой адресация первого операнда определяется в режиме индексирования, а второго — в автоинкрементном.

3.3.4. Команды переходов по условию

Команды переходов (BEQ, BNE, BPL, BMI и BR), о которых уже говорилось в разд. 2.4, являются однословными и имеют приведенный ниже формат:



Старший байт здесь содержит код операции, младший — смещение в словах (7 бит — смещение, 8 — его знак), которое позволяет вычислить действительный адрес перехода относительно счетчика команд PC.

Команды переходов обеспечивают переход назад на расстояние до 128 слов и вперед до 127 слов. Вычисление адреса перехода осуществляет ассемблер, который «знает», что *во время выполнения команды* CPU вычисляет этот адрес следующим образом:

1) расширяет знак смещения на биты 8—15;

2) умножает результат на 2, чтобы получить необходимое смещение в байтах;

3) добавляет результат вычислений к PC для получения необходимого адреса.

Таким образом, *ассемблер* должен исполнить обратную операцию, а именно сформулировать смещение (в словах) от целевого адреса до ячейки, содержащей команду перехода. Обозначим его через T. Важно помнить, что, когда во время выполнения команды смещение добавляется к PC, последний содержит адрес команды, которая следует за инструкцией перехода; поэтому при вычислении адреса перехода PC надо уменьшить на 2. И тогда если $PC = X + 2$, где «X» есть символический адрес самой команды перехода, то

$$\text{количество слов смещения} = (T - X - 2)/2 \text{ (оставлено 8 бит).}$$

Итак, 8-битовое смещение позволяет осуществлять переход в обратном направлении на 200(8) слов (400(8) байт) от текущего значения PC и в направлении увеличения адресов — на 177(8) слов (376(8) байт) от текущего PC.

В следующем сегменте программы иллюстрируется относительный переход:

```
000100  005203   HERE:  INC R3
                        .
                        .
000150  000753         BR HERE
```

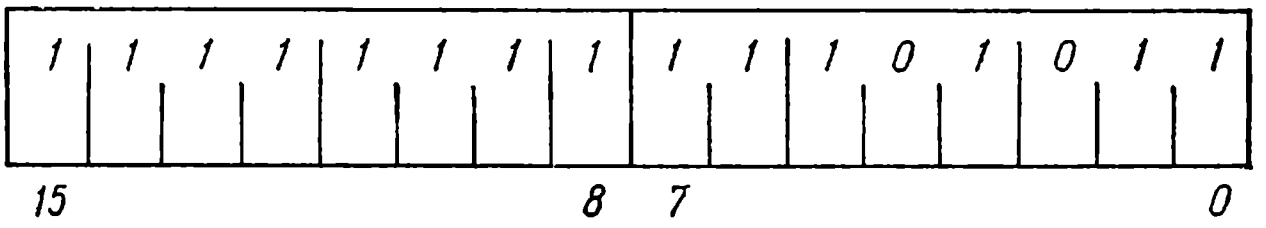
Ассемблер, используя 16-битовую арифметику следующим образом, вычисляет смещение:

```
branch target address   =          100
current location        =          150
                                  ------
difference (100-150)    =       177730
less 2                  =       177726
halved                  =       177753
truncated to 8 bits     =          353
```

и команда перехода имеет вид

000400 + 353 = 000753.

Надо иметь в виду, что когда происходит усечение 16-битового слова до 8-битового байта, то восьмеричное представление числа не совпадает с границами байт. В предыдущем примере 16-битовое число 177753 в двоичном виде представляется как



Значение старшего байта 377, значение младшего байта — число 353. Поэтому мы предостерегаем читателя от простого деления 16-битового слова на две части, ибо получится неверный результат.

3.3.5. Использование PC в качестве общего регистра

Хотя регистр 7 и является общим, он в PDP-11 выполняет также функцию PC. Как только процессор использовал PC для получения слова из памяти (например, при загрузке команды), его значение автоматически увеличивается на 2 для адресации следующего слова выполняемой команды или для указания на следующую, которая должна выполняться. (Когда программа использует PC для доступа к данным, хранящимся в байте, он все равно увеличивается на 2.)

Вообще PC может применяться в любых стандартных режимах адресации PDP-11. Однако имеются четыре из них, использование PC с которыми дает преимущества при работе в случае *позиционно-независимого* кода (см. гл. 4) и неструктурированных

данных. Это *непосредственный, абсолютный, относительный* и *косвенно-относительный* режимы. Их перечень приведен в табл. 3.12.

Таблица 3.12. **Суммарная таблица режимов, использующих PC**

Двоичный код	Наименование режима	Обозначение на Ассемблере	Функция
010	Непосредственный	# n	Операнд непосредственно следует за командой.
011	Абсолютный	@ # n	Абсолютный адрес операнда непосредственно следует за командой.
110	Относительный	A	Адрес операнда, вычисленный относительно команды, расположен непосредственно за ней.
111	Косвенно-относительный	@A	Адрес ячейки, содержащей адрес операнда, определенный относительно команды, непосредственно следует за ней.

Читатель должен помнить, что эти четыре режима выполняются почти так же, как аналогичные, описанные ранее; разница лишь в том, что в них используется общий регистр номер 7 (R7), одновременно выполняющий функции PC.

Заметим, что в примере (табл. 3.13), демонстрирующем *непосредственный режим* адресации, перед загрузкой и выполнением команды ADD PC указывает на ее первое слово. Процессор загружает его содержимое и увеличивает PC на 2 для адресации второго слова команды. Спецификация операнда-источника, точнее, вычисление его адреса, будет производиться по режиму 2 с регистром 7 (автоинкремент PC), следовательно, PC обеспечивает загрузку первого слова команды, а затем адресует ее второе слово, содержимое которого складывается с содержимым регистра R0. Далее PC увеличивается на 2 и указывает на адрес очередной команды.

Таблица 3.13. **Непосредственный режим адресации (режим 2, регистр 7)**

Символьный вид	Восьмеричный код команды	Описание
ADD # 10, R0	062700 000010	Константа «10», находящаяся во 2-м слове команды, добавляется к R0.
ДО		ПОСЛЕ
память (001000) = 062700 (001002) = 000010	регистр (R0) = 000020 (PC) = 001000	память (001000) = 062700 (001002) = 000010
		регистр (R0) = 000030 (PC) = 001004

Абсолютный режим адресации (табл. 3.14) эквивалентен косвенно-непосредственному, т. е. автоинкрементному косвенному режиму с использованием PC. Здесь содержимое второго слова команды берется как адрес операнда. Таким образом, выполнение команды происходит почти так же, как и при непосредственном режиме адресации. Сходство состоит в следующем:

1) PC увеличивается на 2, чтобы указывать на непосредственно следующие данные. Этот шаг считается частью *цикла загрузки команды;*

2) PC снова увеличивается на 2, чтобы указывать на очередную команду, теперь уже *как результат выполнения автоинкрементного режима.*

Единственная разница состоит в том, что данные второго слова интерпретируются не как сам операнд, а как его абсолютный адрес.

Таблица 3.14. **Абсолютный режим адресации (режим 3, регистр 7)**

Символьный вид	Восьмеричный код команды	Описание
CLR @# 1100	005037 001100	Очистить ячейку 1100
ДО память (001000) = 005037 (001002) = 001100 . . (001100) = 012345	регистр (PC) = 001000	ПОСЛЕ память (001000) = 005037 (001002) = 001100 . . (001100) = 000000
		регистр (PC) = 001004

Заметим, что в примерах непосредственной и абсолютной адресации если «n» есть некоторая константа, то #n — ее числовое выражение. Если «n» — символическое имя, то #n обозначает *значение* этого имени. Такие соглашения приняты в Ассемблере. Если символическое имя суть *метка*, то во время ассемблирования ей будет присвоено значение счетчика адресов, достигнутое в момент появления метки. Так, например, команда MOV #X, Rn может быть применена для загрузки некоторого числа X (возможно, *адреса*) в Rn, в то время как MOV @#X, Rn загружает *содержимое* ячейки, имеющей метку X, в Rn. Альтернативный способ спецификации *содержимого* некоторой ячейки с помощью символического адреса заключается в использовании относительного режима адресации.

Относительный режим (табл. 3.15) ассемблируется как индексный с применением регистра R7. База для вычисления адреса, которая помещена во второе или третье слово команды, добавляется к PC, образуя адрес операнда.

Таблица 3.15. **Относительный режим адресации (режим 6, регистр 7)**

Символьный вид	Восьмеричный код команды	Описание
INC A	005267 000074	Содержимое ячейки памяти, непосредственно следующей за командой, добавляется к PC, образуя адрес A. Значение A увеличивается на 1.
ДО память (001000) = 005267 (001002) = 000074 . . (001100) = 001200 *Примечание*: 1004 + 74 = 1100	регистр (PC) = 001000	ПОСЛЕ память (001000) = 005267 (001002) = 000074 . . (001100) = 001201	регистр (PC) = 001004

Косвенно-относительный режим (табл. 3.16) подобен предыдущему, за тем исключением, что второе слово команды, будучи добавленным к PC, образует не адрес операнда, а адрес адреса операнда.

Таблица 3.16. **Относительный косвенный режим адресации (режим 7, регистр 7)**

Символьный вид	Восьмеричный код команды	Описание
CLR @A	005077 000020	Добавить 2-е слово команды к PC, образуя адрес A, по которому находится адрес операнда. Ячейку операнда очистить.
ДО память (001000) = 005077 (001002) = 000020 . . (001024) = 010100 . . (010100) = 135531 *Примечание*: 1004 + 20 = 1024	регистр (PC) = 001000	ПОСЛЕ память (001000) = 005077 (001002) = 000020 . . (001024) = 010100 . . (010100) = 000000	регистр (PC) = 001004

Заслуживает внимания следующая особенность, касающаяся абсолютной и относительной адресации. Считается, что на PDP-11

при обозначении адресов символическими именами обычно используется *относительный режим* адресации. И это соответствует действительности, так как одно- и двухоперандные команды *должны* ссылаться на общий регистр. И когда программист записывает, например,

```
HERE:  MOV X,R3
```

то, *база*, на основании которой будет вычисляться исполнительный адрес во время выполнения, не равна адресу операнда «X». Напротив, это то относительное число, которое, будучи добавленным к PC, обеспечит получение такого адреса. Таким образом, база, которая хранится во втором слое команды, равна X — PC, где X равно действительному адресу. Разберем подробнее этот режим адресации.

Если команда MOV X, R3 находится в ячейке 20, а символическая переменная X хранится в ячейке 100, то мы будем иметь

```
(20)= 016703
(22)= 000054
```

Во время выполнения процессор загружает команду MOV и добавляет 2 к PC, который теперь будет указывать на ячейку 22. Адрес операнда-источника специфицирован режимом 6 с использованием регистра 7. Поэтому процессор загружает слово, на которое указывает PC, и снова увеличивает содержимое PC на 2. Теперь PC указывает на ячейку памяти с адресом 24. И адрес операнда-источника вычисляется следующим образом: к базе прибавляется содержимое выбранного регистра, т. е. PC, образуя адрес операнда:

$$\text{база} + R7 = 54 + 24 = 100, \text{ адрес операнда}$$

Так как ассемблер присваивает метке «HERE» значение адреса первого слова команды, для вычисления адреса операнда-источника можно указать эквивалентное предложение:

```
HERE:  MOV X-HERE-4(PC),R3
```

Ассемблер «знает», как вычислять символические адреса. Мы же для большего понимания приведем две эквивалентные команды, которые также *одинаково ассемблируются.*

```
HERE:  ADD  X,Y
```

```
HERE:  ADD X-HERE-4(PC),Y-HERE-6(PC)
```

Рассмотренный режим называется относительным потому, что адрес операнда вычисляется относительно текущего значения PC. Для обозначения этого применяется база — расстояние (в байтах) между операндом и текущим PC. Если оператор и операнд,

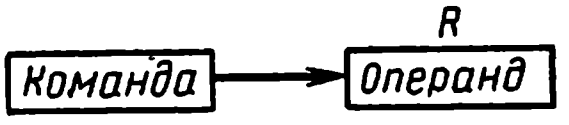
на который он действует, перемещать в памяти так, что расстояние между ними не изменяется, то с точки зрения адресации такая команда всегда будет работать правильно. (Команды переходов по определению являются командами с относительной адресацией и поэтому могут быть использованы для составления позиционно-независимых кодов.)

Так как адрес операнда выражается некоторым значением по отношению к текущему PC, написание программы, динамически вычисляющей подобные смещения во время выполнения, весьма сложная задача. Следовательно, на машинах типа PDP-11, где обычным является относительный режим адресации, рекомендуется избегать составления самомодифицирующихся программ. Сущность применения операндов с относительной адресацией излагается в гл. 4. Там же обсуждается программирование в позиционно-независимых кодах.

Ниже приводятся диаграммы, графически демонстрирующие действие обсуждавшихся режимов адресации.

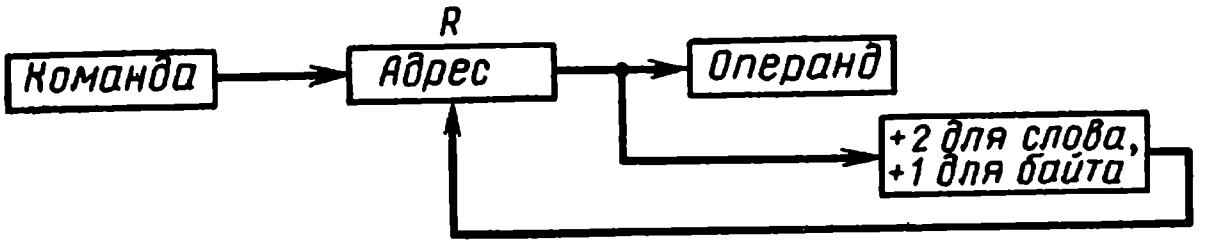
Режимы адресации с использованием общего регистра

Режим 0, регистровый OPR R [R содержит операнд.]

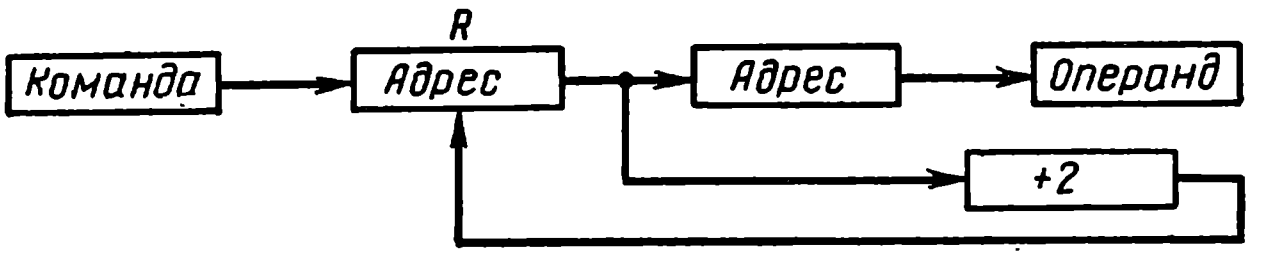


Режим 1, косвенно-регистровый. OPR R или @R [R содержит адрес операнда.]

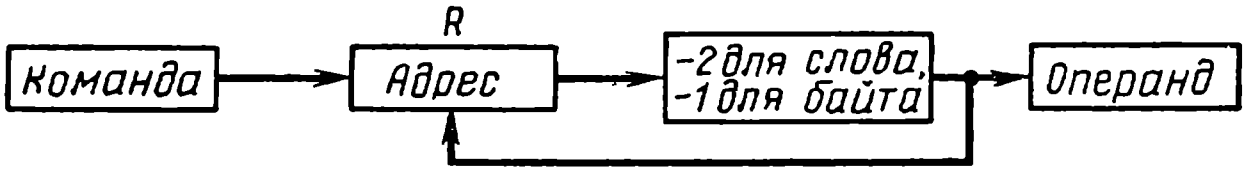
Режим 2, автоинкрементный. OPR (R) + [R содержит адрес операнда, после выполнения команды происходит увеличение содержимого регистра R.]



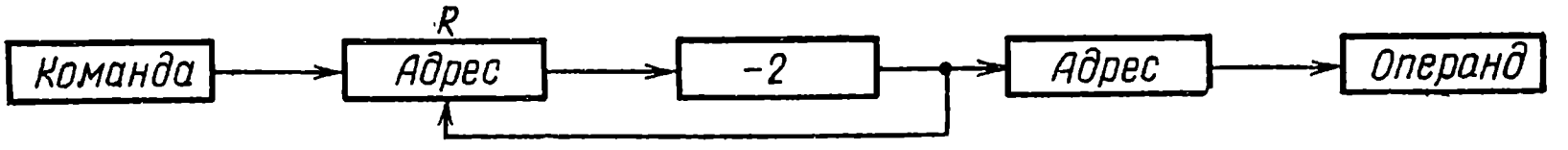
Режим 3. Косвенный автоинкрементный. OPR @ (R) + [R содержит адрес адреса операнда, затем происходит увеличение R на 2.]



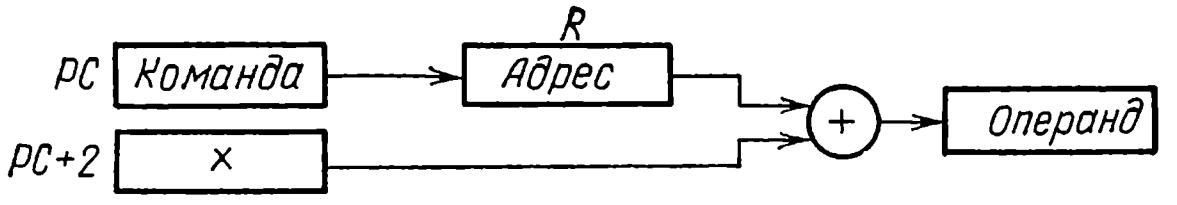
Режим 4, автодекрементный. OPR — (R) [Сначала уменьшение R, затем R содержит адрес операнда.]



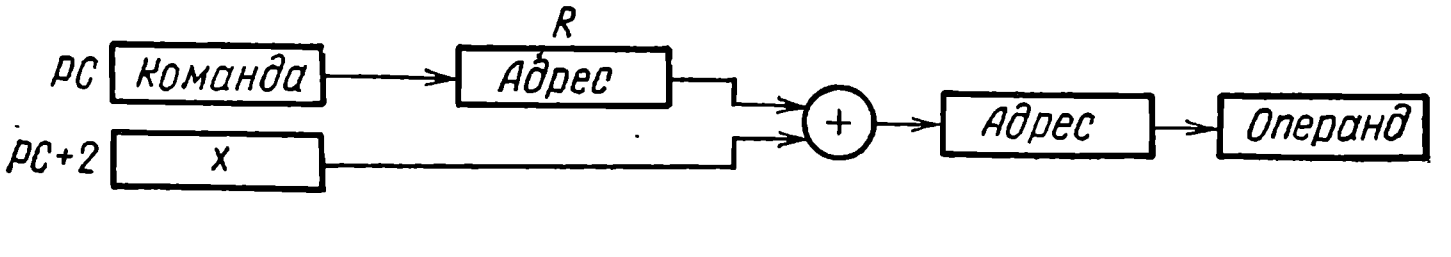
Режим 5, косвенный автодекрементный. OPR @ — (R) [Сначала уменьшение R на 2, затем R содержит адрес адреса операнда.]



Режим 6, индексный. OPR X(R) [Адрес операнда получается из R плюс X, X — во втором слове команды.]



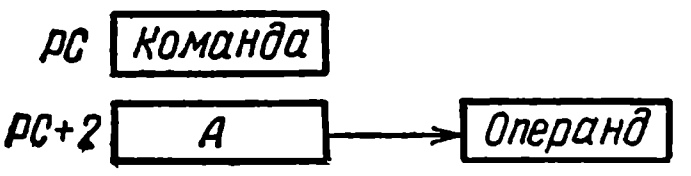
Режим 7, косвенный индексный. OPR @X(R) [Адрес адреса операнда получается R плюс X, X — во втором слове команды.]



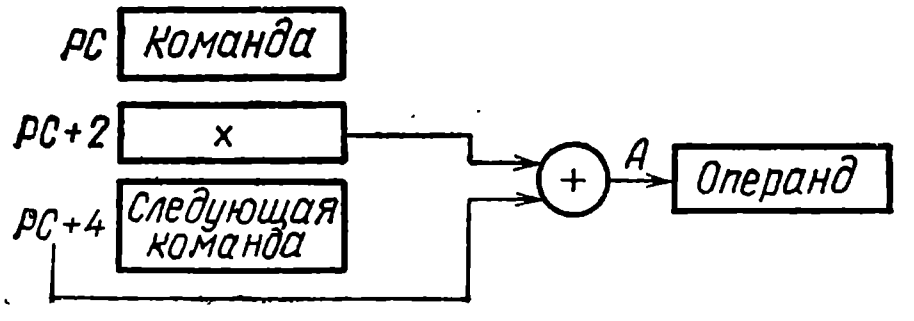
Адресные режимы с использованием счетчика команд

Режим 2, регистр 7; непосредственный. OPR #*n* [Второе слово команды является операндом.]

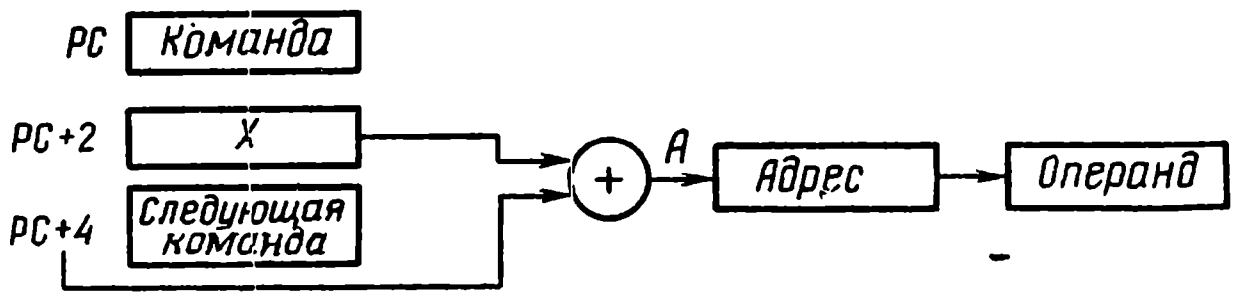
Режим 3, регистр 7; абсолютный. OPR @#A [Второе слово команды есть адрес операнда A.]



Режим 6, регистр 7; относительный. OPR A [Адрес операнда A — сумма X и PC + 4. X находится во втором слове команды.]



Режим 7, регистр 7; косвенно-относительный. OPR @A [Вычисляется адрес А как сумма PC + 4 и X, X — во втором слове команды. По адресу А находится адрес операнда.]



3.4. АССЕМБЛЕР MACRO-11

Изучение различных форм адресации машины PDP-11 облегчит дальнейшее освоение материала. Поэтому полезно обсудить символическую программу совместно с ее восьмеричным кодом. Но сначала мы должны рассмотреть несколько характеристик MACRO-11, которые расширяют общие концепции Ассемблера, обсуждавшиеся в разд. 2.6.

В MACRO-11 точка (.) есть символ, которым обозначается счетчик адреса ассемблера (LC). Когда точка стоит в поле операнда команды, она означает адрес первого ее слова; когда же точка стоит в поле операнда директивы Ассемблера, она означает адрес текущего слова или байта. Например:

```
A:      MOV     #.,R0   ;"." REFERS TO LOCATION A; I.E, THE
                        ; ADDRESS OF THE  "MOV" INSTRUCTION
```

Заметим, что если «#» читается как *некоторое значение* и «.» означает «*здесь*», то «#.» воспринимается как *...значение этой ячейки...* или, проще, «адрес ячейки», «*этот адрес*».

В начале каждого прохода трансляции ассемблер устанавливает счетчик адреса в нуль. Обычно каждому байту/слову генерируемого объектного кода назначаются последовательные ячейки памяти. Однако та из них, куда помещается объектный код, может получить конкретное значение путем принудительной установки счетчика адреса.

Такое *предложение прямого присваивания* связывает символ с его значением. И если присваивание делается первый раз, то символ вместе со своим значением заносится в таблицу символов Ассемблера. Впоследствии этот символ можно переопределять, присваивая ему новое значение, которое заменит в таблице прежнее. Общий формат предложения для прямого присваивания имеет вид

символ = выражение

Одно из применений прямого присваивания — задание начального значения счетчику адреса (например, .=1000); другое — присвоение символическим именам значений наиболее часто употребляемых общих величин и выражений, что увеличивает информационное содержание программы (и, следовательно, ее ясность).

Десятичная точка интерпретируется Ассемблером по-разному. Будучи использованной самостоятельно, она представляет значение LC и, следовательно, может быть использована для инициализации LC (например, .=1000), а также в качестве операнда команды. Например, команда BR .—4 вызывает переход на инструкцию, которая хранится по адресу на два слова меньше, чем BR.

Общая ошибка — это запись команды BR .—2 с целью вернуться на две инструкции назад. Она состоит в том, что (1) в PDP-11 можно адресоваться как к половине, так и к целому слову, (2) инструкции для PDP-11 могут занимать одно, два или три слова. Таким образом, использование «.» для вычисления адреса перехода бессмысленно, если программист не знает, сколько слов или байт надо обойти. Следовательно, применение меток, *точно* указывающих операнд,— надежная гарантия избежать ошибок.

Второе применение десятичной точки связано с *директивами Ассемблера,* или псевдооперациями, которые управляют процессом ассемблирования и могут генерировать данные. В качестве примеров псевдоопераций можно назвать ранее обсуждавшиеся директивы .END и .WORD.

Директива .WORD может иметь один или несколько операндов. Для каждого из них в объектной программе отводится одно слово. Если в директиве имеется более чем один операнд, то они будут храниться в последовательных словах памяти. В качестве операнда может быть использовано любое правильно оформленное выражение. *Если в предложении отсутствует код операции, а в поле операндов имеется одно или более выражений, то такое предложение интерпретируется, как директива .WORD. В при*мере последовательные ячейки памяти хранят значения 5, 6, 7 и 8.

```
FIVE:   5
SIX:    .WORD   6
SEV:    .WORD   7,8.
```

Наконец, третий случай применения «.» (точки). MACRO-11 считает, что все числа восьмеричные, если нет специальных указаний о другой системе счисления. Так вот, «.» используется для информирования ассемблера о том, что число является десятичным. Перечисленные символы и соглашения будут применяться ниже, при кодировании программ на MAKRO-11. Заметим, что все приведенные примеры являются реальными программами для PDP-11 (или их сегментами) и могут быть ассемблированы и выполнены на машинах PDP-11.

3.4.1. Примеры программ

Рассмотрим проблему массива из гл. 2, блок-схема которой изображена на рис. 2.9а. Если не учитывать переменной длины команд реальной PDP-11, то можно предположить, что каждая

из них требует одного слова и последовательные команды занимают последовательные ячейки памяти. Но, как следует из предыдущего материала, такое предположение *ошибочно* и фактический ассемблерный листинг программы выглядит так, как это показано на рис. 3.3. Из него очевидны два тонких изменения по сравнению с программой, написанной для гипотетической ЭВМ (см. гл. 2).

Первое — это то, что команда с меткой FIXUP здесь модифицируется прибавлением двух к ее второму слову (одно слово этой команды занимает код операции, следующее — операнд-источник и третье — операнд-приемник). Константа «два» должна быть прибавлена к содержимому второго слова, так как оно содержит относительный базовый адрес массива K.

```
Адрес команды  Содержимое ячейки  Метка    Код операции  Операнд(ы)        Комментарии
1              000000              .ASECT                                   ;ASSEMBLE ABSOLUTE CODE
2              001000              .=1000                                   ;STARTING ADDRESS = 1000
3  001000      016767  START:      MOV       ZERO,ISUM                      ;INITIALIZE SUM
               000032
               000032
4  001006      066767  FIXUP:      ADD       K,ISUM                         ;ADD IN ELEMENT OF ARRAY K
               000034
               000024
5  001014      026767              CMP       FIXUP+2,LIMIT                  ;DONE YET
               177770
               000022
6  001022      001404              BEQ       STOP                           ;YES
7  001024      066767              ADD       CONS,FIXUP+2                   ;MODIFY ADDRESS
               000012
               177756
8  001032      000765              BR        FIXUP                          ;LOOP
9  001034      000000  STOP:       HALT                                     ;STOP HERE
10 01036       000000  ZERO:       .WORD     0                              ;CONSTANT ZERO
11 01040       000000  ISUM:       .WORD     0                              ;RESULTANT ZERO
12 01042       000002  CONS:       .WORD     2                              ;CONSTANT TO GENERATE
                                                                            ;NEXT ADDRESS
13 01044       000176  LIMIT:      K-FIXUP-4+98.                            ;LAST RELATIVE ADDRESS
                                                                            ; GENERATED
14 01046               K:          .BLKW     50.                            ;RESERVE 50 LOCATIONS
                                                                            ; FOR ARRAY K
15             001000'             .END      START
```

Таблица символов

```
CONS    001042          FIXUP   001006          ISUM    001040
K       001046          LIMIT   001044          START   001000
STOP    001034          ZERO    001036
```

Рис. 3.3. Программа сложения чисел на PDP-11, применяющая модификацию команд

Второе изменение состоит в том, что слово, помеченное меткой LIMIT, содержит ссылку на адрес последнего элемента массива K. Эта величина равна K—FIXUP—4 + 98., где K—FIXUP—4 представляет собой первоначальный относительный адрес, а десятичное 98 — количество элементов массива, уменьшенное на единицу (т. е. оно определено как 2 * [N —1). Заметим, далее, что когда команда CMP сравнивает две адресные величины, то она оперирует их числовыми значениями.

Напомним читателю, что хотя самомодифицирующиеся программы и бывают весьма эффективны, однако они не рекомен-

дуются для практики программирования. Приведенная выше (см. рис. 3.3) самомодифицирующаяся программа не может быть повторно запущена на выполнение без переустановки второго слова инструкции, помеченной меткой FIXUP. И инициализирующее

```
MOV     #K,FIXUP+2
```

предложение не восстановит начального значения второго слова команды, так как здесь использован относительный режим адресации. Цель данного примера — показать, как ассемблируются

```
Адрес команды  Содержимое ячейки  Метка  Код операции  Операнд(ы)   Комментарии
1                                 ;THIS IS A PROGRAM TO SUM THE ELEMENTS OF THE ARRAY "K'".
2                                 ;THE SUM IS LEFT IN R0.
3              000000                 .ASECT
4              000000                 R0=%0                 ;DEFINE REGISTER 0 AS SYMBOL R0
5              000001                 R1=%1                 ;DEFINE REGISTER 1 AS SYMBOL R1
6              001000                 .=1000                ;DEFINE STARTING ADDRESS
7 001000       005000 START:          CLR     R0            ;INITIALIZE SUM
8 001002       012701                 MOV     #K,R1         ;ADDRESS OF K MOVED INTO R1
               001020
9 001006                LOOP:
10 01006       062100                 ADD     (R1)+,R0      ;ADD IN VALUE
11 01010       022701                 CMP     #K+18.,R1     ;DONE?
               001042
12 01014       100374                 BPL     LOOP          ;NO, CONTINUE
13 01016       000000                 HALT
14 01020       000001 K:              1,2,3,4,5,6,7,8.,9.,10.  ;NOTE THAT ".WORD"
   01022       000002                                          ;IS IMPLIED
   01024       000003
   01026       000004
   01030       000005
   01032       000006
   01034       000007
   01036       000010
   01040       000011
   01042       000012
15             001000'                .END    START
Таблица символов
K          001020                 LOOP    001006          R0     =%000000
R1       =%000001                 START   001000
```

Рис. 3.4. Программа суммирования элементов массива, применяющая автоинкрементный режим PDP-11

реальные программы для PDP-11 и как выглядит их объектный код.

На рис. 3.4 приведен листинг упрощенной версии программы, решающей ту же проблему массива. Для краткости размер массива был уменьшен до 10 элементов. Тут иллюстрируется также применение операции присвоения некоторого значения символу. Так, например, «%0» (в ассемблере это обозначает регистр) присваивается символу «R0».

3.5. СЕМЕЙСТВО PDP-11 И ИЕРАРХИЯ ПАМЯТИ

В течение нескольких последних лет резко уменьшилась стоимость аппаратуры систем. Производство CPU на чипах привело

к появлению одноплатных вычислительных систем, имеющих различные типы памяти (память для чтения/записи (RAM), память только для чтения (ROM)), а также интерфейсы параллельного и последовательного ввода-вывода. Причем стоимость подобной системы такова, что десять лет назад на эту сумму можно было приобрести только один из интерфейсов ввода-вывода. Однако стоимость программного обеспечения, разработка которого требует значительных трудозатрат, продолжает возрастать. Применение языков высокого уровня позволяет достигнуть высокой совместимости программ; тем не менее максимальную эффективность имеют те из них, которые полностью учитывают особенности данной архитектуры.

3.5.1. Понятие семейства машин

Идея производства семейства CPU с различными возможностями при сохранении общей архитектуры, но по-разному реализованных не нова. Цель такого производства — совместимость программного обеспечения всех CPU на *уровне машинного языка*. Выгода от применения такой стратегии огромна, поскольку, несмотря на постоянное развитие программного обеспечения, оно в любое время может использоваться на ранних моделях семейства. При этом оно легко модифицируется для учета архитектурных отличий и других особенностей конкретной машины. Кроме того, существование семейства ЭВМ допускает простую замену одного CPU на другое, разработанное на базе вновь появившейся технологии. Это, в свою очередь, позволяет ориентироваться на новейшее программное обеспечение. Наблюдающееся к тому же влияние более новой технологии памяти (кэш-память, полупроводниковая повышенной емкости, полупроводниковая с имитацией дисковой организации данных) может повлиять на создание новых программных средств. Развивается и системная периферия, что дает возможность улучшить использование существующих CPU. И наконец, общая архитектура, ориентированная на пользователя, позволяет обществу DECUS (Digital Equipment Computer User's Society — для PDP-11) — ассоциации пользователей — создавать огромное количество полезных программ.

3.5.2. Технологическое использование

Понятие семейства лучше всего демонстрируют машины PDP-11 фирмы DEC, являющиеся самыми распространенными на сегодняшний день мини-ЭВМ. С момента появления PDP-11 в 1970 г. к настоящему времени разработано около 10 их различных моделей, которые можно сгруппировать в три хронологических «потока».

1. *Постоянная стоимость, совершенствующееся исполнение*: 11/20 (1970), 11/40 (1972), 11/60 (1977).

2. *Уменьшающаяся стоимость, постоянная технология исполнения*: серии (i) — 11/20 (1970), 11/05,10 (1973), 11/04,03 (1975), несколько PDT-11 (1978); серии (ii) — 11/40 (1972), 11/34 (1976), 11/23,24 (1979).

3. *Растущая стоимость, существенно совершенствующееся исполнение*: 11/20 (1970), 11/45,55 (1972, 1976), 11/70 (1974), VAX-11/780 (1977).

Под словом «стоимость» подразумевается минимальная цена системы, а под «исполнением» — ее функциональность, в том числе скорость и надежность работы процессора. Улучшение исполнения может быть достигнуто за счет увеличения скорости выполнения операций CPU, расширения архитектуры (например, включение процессора плавающей точки) или улучшения исполнения памяти (например, кэш-память). В семействе машин PDP-11 были выполнены следующие улучшения:

1. *Расширенный арифметический элемент* (EAE). Периферийное устройство, присоединенное к UNIBUS и обеспечивающее умножение /деление/ сдвиг чисел с фиксированной точкой.

2. *Расширенный набор команд* (EIS). Архитектурное улучшение CPU, использующее общие регистры для выполнения операций умножения /деления/ сдвига над числами с фиксированной точкой.

3. *Процессор плавающей точки* (FPP). Набор команд арифметики с плавающей точкой и 6 регистров, обеспечивающих выполнение операций.

4. *Быстрая шина* (FASTBUS) *и ассоциативная высокоскоростная память.* Быстрая шина подсоединяет CPU к одному из двух каналов биполярной памяти, к другому присоединена UNIBUS.

5. *Набор команд плавающей точки* (FIS). Набор команд плавающей точки, ориентированный на работу с помощью стека (т. е. нулевой адрес).

6. *Кэш-память.* Высокоскоростная биполярная память, вставленная между CPU и основной памятью, и соответствующие устройства управления.

7. *Диспетчер памяти.* Устройство отображения/защиты, расширяющее физическое адресное пространство.

8. *MASSBUS.* Шина, предназначенная для обеспечения высокой скорости ввода-вывода

3.5.3. Совместимость

Машины PDP-11 архитектурно совместимы снизу вверх, что позволяет использовать на них одно и тоже программное обеспечение и оборудование. UNIBUS допускает применение оборудования одной машины на другой. В приведенной ниже таблице суммированы возможности оборудования для различных машин PDP-11:

Семейство машин PDP-11

Свойства	11/03	11/04	11/05, 10	11/31	11/35, 40	11/60	11/45, 70
Общие регистры	8	8	8	8/9	8/9	8/9	16
Диспетчер памяти	Нет	Нет	Нет	Есть	Есть (О)	Есть	Есть (О)
Автоматическая приоритетность прерываний	Однолинейная, многоуровневая	Многолинейная, многоуровневая	Многолинейная, многоуровневая	Многолинейная, многоуровневая	Многолинейная, многоуровневая	Многолинейная, многоуровневая	Многолинейная, многоуровневая
Режимы процессора	1	1	1	2	2	2	3
Расширенная арифметика	EIS (О) FIS (О)	EAE (О)	EAE (О)	EIS FPP (О)	EIS (О) FIS (О)	EIS FPP	EIS FPP (О)
Кэш-память	Нет	Нет	Нет	Есть (О)	Есть (OEM)	Есть	Нет (45) Есть (70)
Микропрограммирование	Есть (О)	Нет	Нет	Нет	Есть (OEM)	Есть (О)	Нет
Сложение регистр—регистр	3,5 мкс	3,2 мкс	3,7 мкс	2,1 мкс	1,0 мкс	0,34 мкс	0,30 мкс

(О) означает, что данная возможность определяется фирмой DEC, а (OEM) — что она может определяться другой фирмой-изготовителем. Время выполнения операции сложения типа регистр — регистр (в микросекундах) весьма показательно для характеристики исполнения CPU. Заметим, что компиляторы Fortran IV позволяют пользователю специфицировать необходимое оборудование в период компиляции (F IV, версия II) или в период компоновки программы (F IV, версия I). Компилятор Fortran IV-PLUS требует наличия команд процессора плавающей точки и связанных с ними регистров.

3.5.4. Фазы обработки команды

Теперь полезно дать краткое описание основных этапов выполнения любой специфичной для PDP-11 команды и посмотреть, как они влияют на общее время ее выполнения. Для этого проведем детальный анализ *I*-, *O*- и *E*-фаз обработки команды PDP-11 на примере *двухоперандной команды,* используя PC, MAR и MDR-регистры, которые, как указывалось, имеются на

всех ЭВМ. Аналогичный анализ для PDP-11/60 будет проведен позже (в разд. 6.10).

Выборка команды (*I*-фаза): *I*-фаза в РДР-11 отличается от описанных ранее тем, что PC увеличивается на 2 после выборки команды:

```
1. (MAR) ← (PC)          ;ADDRESS OF INSTRUCTION INTO MAR
2. (MDR) ← ((MAR))       ;FETCH FIRST WORD OF INSTRUCTION
3. (PC)  ← (PC) + 2      ;INCREMENT PC
4. (IR)  ← (MDR)         ;COPY MDR INTO IR
```

В *O*-фазе производится обработка как операнда-источника, так и операнда-приемника двухоперандной команды. Если положить, что Rs и Rd есть внутренние регистры, которые содержат исходный *операнд* и *адрес* назначения соответственно, то *O*-фаза может быть описана следующим образом.

***O*-фаза:** сначала выбирается операнд-источник:

```
If IR[11:10] = 0, then  (Rs)  ← (Rn), n = IR[8:6]        ;REGISTER MODE

If IR[11:10] = 1, then  (MAR) ← (Rn), n = IR[8:6]        ;AUTOINCREMENT MODE
                        (MDR) ← ((MAR))                  ;MEMORY FETCH
                        (Rs)  ← (MDR)
                        (Rn)  ← (Rn) + 2                 ;INCREMENT Rn

If IR[11:10] = 2, then  (Rn)  ← (Rn) - 2, n = IR[8:6]    ;AUTODECREMENT MODE
                        (MAR) ← (Rn)
                        (MDR) ← ((MAR))                  ;MEMORY FETCH
                        (Rs)  ← (MDR)

If IR[11:10] = 3, then  (MAR) ← (PC)                     ;INDEX MODE
                        (PC)  ← (PC) + 2                 ;INCREMENT PC
                        (MDR) ← ((MAR))                  ;FETCH INDEX WORD
                        (MAR) ← (MDR) + (Rn), n = IR[8:6] ;COMPUTE ADDRESS
                        (MDR) ← ((MAR))
                        (Rs)  ← (MDR)

If IR[9] = 1,     then  (MAR) ← (Rs)                     ;DEFERRED ADDRESSING
                        (MDR) ← ((MAR))                  ;FETCH SOURCE OPERAND
                        (Rs)  ← (MDR)
```

После этого он запоминается в Rs, и далее выполняется вторая половина *O*-фазы — работа с операндом-приемником:

```
If IR[5:4] = 0,   then (Rd)  ← 777700 + n, n = IR[2:0] ;COMPUTE ADDRESS OF Rn

If IR[5:4] = 1,   then (Rd)  ← (Rn), n = IR[2:0]       ;AUTOINCREMENT MODE
                       (Rn)  ← (Rn) + 2

If IR[5:4] = 2,   then (Rn)  ← (Rn) - 2, n = IR[2:0]   ;AUTODECREMENT MODE
                       (Rd)  ← (Rn)

If IR[5:4] = 3,   then (MAR) ← (PC)                    ;INDEX MODE
                       (PC)  ← (PC) + 2                ;INCREMENT PC
                       (MDR) ← ((MAR))                 ;FETCH INDEX WORD
                       (Rd)  ← (MDR) + (Rn), n = IR[2:0] ;COMPUTE ADDRESS

If   IR[3] = 1,   then (MAR) ← (Rd)                    ;INDIRECT ADDRESSING
                       (MDR) ← ((MAR))
                       (Rd)  ← (MDR)                   ;FINAL ADDRESS IN Rd
```

Фаза выполнения (*E*-фаза): IR [15 : 12] используется для того, чтобы определить, какая из двухоперандных команд должна быть выполнена. При этом в качестве первого операнда берется содержимое Rs, а содержимое Rd используется как адрес второго

операнда. Результат помещается по адресу второго операнда и, таким образом, затирает прежнее содержимое ячейки.

В общем время выполнения двухоперандной арифметической команды складывается из времени реализации следующих ее шагов:

1) выборка команды;
2) выборка операнда-источника;
3) выборка операнда-приемника;
4) выполнение команды;
5) сохранение результата.

Обычно шаги 1 и 4, а также 2, 3 и 5 объединяют. Тогда общее время выполнения команды складывается из *выборки/выполнения, команды,* выборки *источника* и *приемника.*

3.5.5. Скорость выполнения программ UNIBUS, FASTBUS и кэш-память

Ниже приведена таблица выполнения команды ADD на PDP-11/45,55 для различных режимов адресации при использовании биполярной (Bp), металлооксидной полупроводниковой (MOS) и магнитной памяти (Cr). В таблице приводится также число циклов памяти (MC), требуемое для соответствующей операции:

Время (микросек) загрузки/выполнения

	SRC MODE 0 DST MODE 0				SRC MODE 1-7 DST MODE 0				SRC MODE 0-7 DST MODE 1-7			
	Bp	MOS	Cr	MC	Bp	MOS	Cr	MC	Bp	MOS	Cr	MC
ADD	.30	.51	.97	1	.45	.66	1.12	2	.75	1.17	1.8	2

Время (микросек) SRC или DST

Mode	Bp	MOS	Cr	MC
0	0	0	0	0
1	.30	.51	.89	1
2	.30	.51	.89	1
3	.75	1.17	1.92	2
4	.45	.66	1.04	1
5	.90	1.32	2.07	2
6	.60	1.02	1.86	2
7	1.05	1.68	2.89	3

Заметим, что биполярная, MOS и память на магнитных сердечниках имеют время циклов приблизительно 0,3; 0,51 и 0,97 мкс соответственно. Как отмечалось ранее, биполярная память и MOS соединяются с CPU через FASTBUS, в то время как память на магнитных сердечниках является периферией UNIBUS. При этом можно сделать следующие общие выводы:

1. Для выборки SRC и DST при режиме адресации 0 требуется время, равное одному циклу памяти.

2. Если режим адресации SRC не нуль, то требуется дополнительное время для оценки адреса SRC, а также для выборки операнда-источника.

3. В аналогичной ситуации для DST может потребоваться предварительное вычисление адреса операнда-приемника, а затем выборка операнда-приемника, выполнение операции ADD и сохранение результата по адресу операнда-приемника. Итак (для данного примера), общее время выполнения существенно пропорционально числу и скорости циклов памяти, требуемых для выборки команды и (если необходимо) содержимого памяти для формирования действительного адреса источника и приемника, их выборки и сохранения результата.

Таким образом, команда ADD R0, R1, хранимая в биполярной памяти, выполняется за 0,3 мкс, в то время как инструкция ADD @X(R5), @ Y(R3) в аналогичном случае, но при использовании магнитной памяти, осуществлялась бы за 7,59 мкс. И наоборот, ADD @ X(R5), @Y(R3) для выполнения в биполярной памяти требует 2,85 мкс, в то время как ADD R0, R1 в магнитной памяти — 0,97 мкс. Очевидно, что для получения максимальной скорости выполнения, например, на PDP-11/45 программист должен быть уверен, что:

1) для хранения промежуточных результатов общие регистры используются настолько полно, насколько это возможно;

2) наиболее критичные в отношении времени программные секции находятся в биполярной памяти, менее критичные — в MOS, а остальные — в памяти на магнитных сердечниках.

Удовлетворение *первого* критерия определяется умением программиста построить структуру алгоритма так, чтобы она полностью учитывала архитектуру машины. Применение микропрограммирования для ускорения выполнения программы будет обсуждаться в гл. 6.

Второй критерий требует анализа активности программ для выделения секций, наиболее критичных по времени выполнения. Далее такие секции средствами *компоновщика* должны быть размещены в соответствующей памяти. Однако подобный подход приемлем лишь по отношению к индивидуальным пользовательским программам. В свою очередь, если всю программу ориентировать на биполярную память, то это будет неэффективно с точки зрения стоимости.

Альтернативным решением этой проблемы можно считать использование нескольких разных типов памяти, представляющих с точки зрения быстродействия некоторую иерархическую структуру. Тогда возможно соответствующее распределение секций программ по различным уровням памяти.

Обычно для организации иерархии памяти служит *кэш-память* — небольшая, быстрая память, расположенная между CPU и относительно медленной, основной (Мр) памятью ЭВМ. Она содержит пары адрес/данные (AD), включает Мр-адреса и копии соответствующих им ячеек основной памяти. Когда CPU обращается к

Мр, то происходит сравнение затребованного адреса с хранимым в кэш-памяти. Если они равны, то CPU выбирает из кэш-памяти соответствующую порцию данных. В противном случае CPU обращается к Мр и новая AD-пара помещается в кэш-память, обычно вытесняя пару, находившуюся в ней до того. В двух килобайтах кэш-памяти (моделей PDP-11/60, 70) для AD-пар хранятся адреса, соответствующие младшим 11 битам AD-пары основной памяти. Таким образом, AD-пары, взятые из ячеек памяти с адресами 0, 2048, 4096 и т. д., будут размещаться, начиная с нулевой ячейки кэш-памяти и вытесняя любую хранившуюся ранее AD-пару. Обычно в программах, особенно в тех, которые содержат циклы, на короткий период времени необходим доступ к небольшим непрерывным группам ячеек основной памяти (Мр). Эти небольшие группы и переносят в кэш-память для того, чтобы большая часть обращений CPU (до 90%) выполнялась из этой быстрой памяти, время доступа к которой обычно составляет лишь небольшую часть от времени доступа к более медленной основной памяти.

Таким образом, кэш-система автоматически и динамически переносит самые нужные программы и данные в небольшую, но быструю память, обеспечивая снижение времени обращения к памяти и, следовательно, высокий темп выборки.

Модель PDP-11/70 имеет подобный PDP-11/45 центральный процессор с кэш-памятью, расположенной между CPU и Мр. В этой машине, когда CPU обращается к Мр, то в 1024-словную кэш-память передается двухсловный блок (содержащий адрес операнда и содержимое предыдущей или последующей ячейки главной памяти), так, чтобы сформировать две прилегающие друг к другу AD-пары. В модели 11/60 размер передаваемого блока равен одному слову. Следовательно, в любом случае *принцип локализации программ* гарантирует резидентность наиболее часто используемых кодов в быстрой памяти. Можно показать, что программы, конструкция которых позволяет не сильно отклоняться от статистического принципа построения кэш-памяти (например, линейное размещение кодов), имеют на модели 11/70 почти такое же время выполнения (95 %), как и при использовании биполярной памяти в машине 11/55.

3.5.6. Расширение памяти: защита и управление

С самого начала конструкторы PDP-11 рассчитали, что 16-битовая мини-ЭВМ с адресным пространством, равным 32 768 слов (32К), не совсем достаточна. Со временем программисты пришли к выводу, что для их программ необходимо больше адресного пространства, и нашли новые способы использования ЭВМ (такие, как мультипрограммирование), которые требуют разделения и защиты программы, одновременно резидентных в памяти.

Первый процессор PDP-11, модель 11/20, генерировал фактически 18-битовые прямые адреса байт, хотя длина слова и опе-

рационная логика в 11/20 была 16-битовой. Таким образом, в машине 11/20 слово позволяет ссылаться на адрес не более 32К, в то время как UNIBUS обеспечивает адресацию 128 Кслов. Это проявляется в том, что обращения к странице ввода-вывода (к адресам между 160000 и 177777) в действительности преобразуются в обращения к диапазону 760000 — 777777, поскольку в 11/20 два бита адреса постоянно установлены в единицу (биты 16 и 17), реализуя указанные ссылки на страницу ввода-вывода.

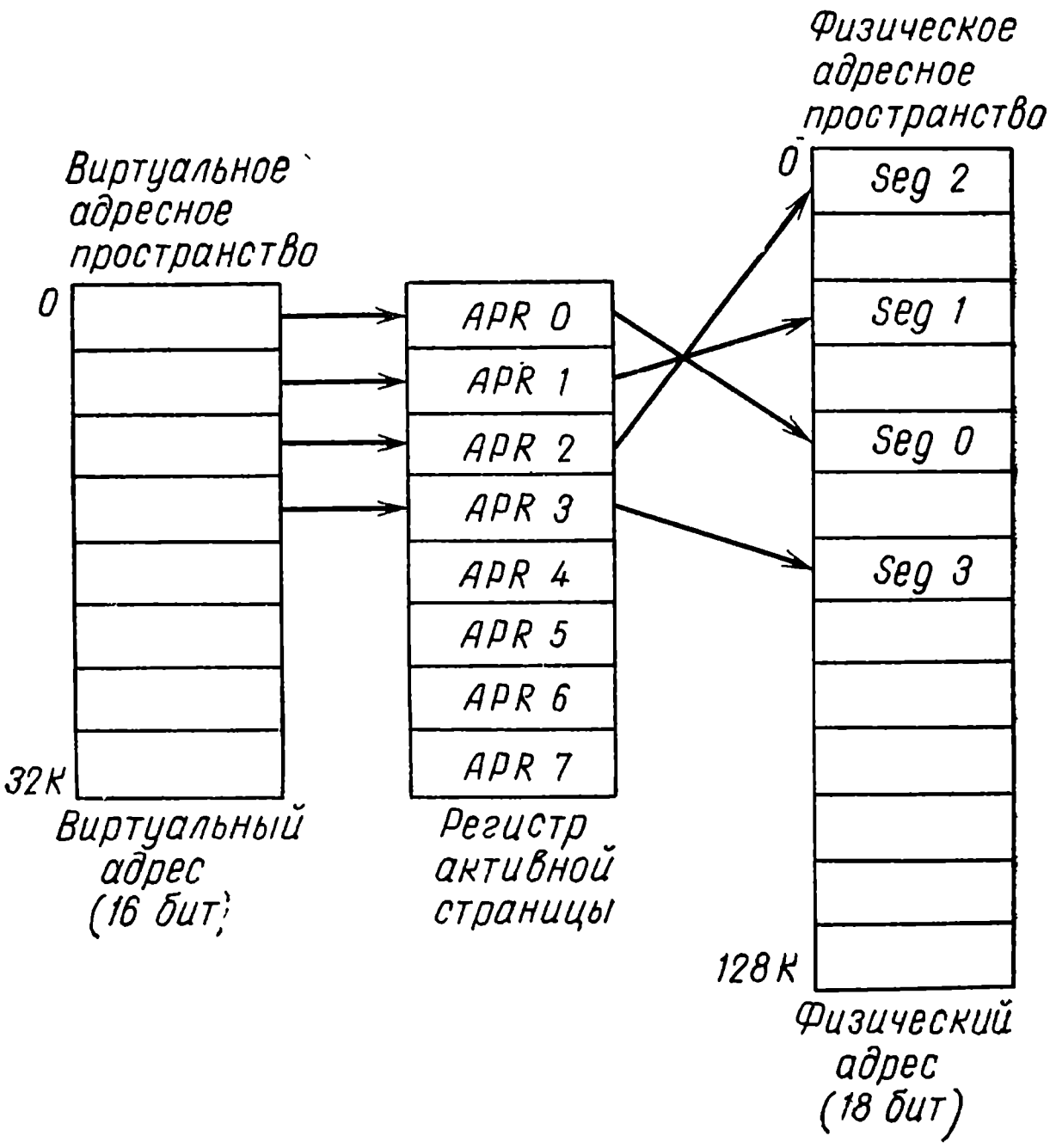


Рис. 3.5. Отображение виртуальных адресов в физические адреса

Со времени ввода в эксплуатацию моделей 11/40 и 11/45 18-битовая адресация стала доступной не только на UNIBUS. Адреса, которые генерируют CPU, теперь также могут быть модифицированы в 18-битовое число. А еще позже, с созданием машины 11/70, появилась возможность иметь 22 бита для задания адреса. Однако остановимся лишь на 18-разрядной схеме, ибо оба механизма адресации весьма близки.

На одних машинах PDP-11 по выбору, на других — в качестве стандартного оборудования применяется диспетчер памяти, позволяющий расширить возможности адресации. Когда это устройство работает, нормальный 16-битовый прямой адрес интерпретируется уже не как физический адрес байта (PA), а как виртуальный (VA), содержащий информацию, применяемую при формировании нового 18-битового физического адреса. Эта

информация комбинируется со смещением и описанием памяти, содержащимися в *регистре активной страницы* (APR), в результате чего и производится 18-битовый PA (рис. 3.5).

Основная задача APR — осуществлять отображение VA на физическое адресное пространство и обеспечивать возможность расширения адресации для систем, имеющих более 28K физической памяти. В моделях 11/34, 35, 40, 60 имеются два набора этих регистров (в моделях 11/45, 55, 70 — больше). Один из этих наборов обеспечивает отображение памяти в режиме ядра, другой — в режиме пользователя. Это позволяет операционной системе иметь одно множество регистров исключительно для собственного применения, в то время как прикладные программы разделяют между собой другое множество. В связи с этим одна из функций многопрограммной операционной системы состоит в загрузке и перегрузке регистров диспетчера памяти при передаче контроля над процессором некоторой задаче пользователя (см. гл. 10).

Каждый APR представляет собой пару 16-битовых регистров: *регистр адреса страницы* (PAR) и *регистр описания страницы* (PDR). PAR содержит начальный адрес страницы в памяти; PDR — информацию о ее размере, о том, как она может быть увеличена, и о режимах доступа к ней.

Виртуальное адресное пространство, используемое программистом, при этом делится на восемь страниц по 4 Кслов. Первые три бита 16-битового виртуального адреса служат для выбора одного из регистров APR. Базовый адрес, который находится в PAR, определяет начало страницы в физическом адресном пространстве.

Например, если виртуальный адрес операнда команды есть 120000, то будет использован APR5, так как первые три бита — это 101. Если APR5 содержит начальный адрес 3200, то действительный физический адрес равен 320000. Более подробно вопросы управления памятью рассматриваются в гл. 7.

УПРАЖНЕНИЯ

1. Предположим, что команда, приведенная ниже, хранится в ячейке 150 и демонстрирует одну из операций MACRO-11. Что изменится в выводимой ассемблером информации, если каждой из констант в инструкции будет предшествовать знак «#»? Пара знаков «@#?

```
        MOV     100,200
```

2. Что произойдет после выполнения следующих последовательностей кодов?

а)
```
        MOV     -(R7),-(R7)
```

б)
```
X:      MOV     (R7),@(R7)+
        BR      X
        HALT
```

в)
```
        MOV     (R7),R0
        MOV     R0,(R7)
```

3. Как поступает ассемблер со следующими парами команд?

```
        .=26
DO:     ADD     R1,R2
        .=100
        BMI     DO
```

4. Используя форматы режимов адресации, приведенные ранее в этой главе, покажите, как команды табл. 3.17 изменят содержимое определенных ячеек памяти (рассмотреть адресное пространство ДО и ПОСЛЕ выполнения). Принять, что каждая команда начинается в ячейке памяти 1000 и начальное содержимое регистров и памяти одинаковы.

Таблица 3.17

Команды	*Начальные значения регистров и ячеек памяти*
a. CLR (R5)+	(R0) = 0, (R2) = 200, (R3) = 1000
b. ADD (R2)+,R4	(R4) = 400, (R5) = 30000
c. ADD -(R3),R0	(A) = 2000, (B) = 500
d. ADD 30(R2),20(R5)	(200) = 1000, (230) = 30000
e. DEC @(R2)+	(500) = 3000
f. ADD @1000(R2),R1	(776) = 100
g. MOV #10,R0	(1000) = 230
h. ADD @#1200,R5	(1200) = 30020
i. NEG A	(3000) = 123456
j. SWAB @B	(30020) = 1000

5. Перепишите задачу о массиве, данную в гл. 2, используя общие регистры для индексирования элементов массива.

6. Многие машины имеют избыточную систему команд. Так, например, функции CLR могут быть выполнены командой MOV. На PDP-11 команда ADD может быть заменена соответствующей последовательностью инструкций. Напишите такую последовательность.

7. Объясните, почему ссылка вперед ($\cdot = A - 6$) в программном сегменте, приведенном ниже, представляет проблему для ассемблера MACRO-11.

```
        .=A-6
        .WORD   0
START:  MOV     #3,R0
A:      CMP     R0,R1
        .
        .
```

8. Используют ли команды BR и JMP смещения в байтах при относительном режиме адресации? В чем при этом состоят разница и сходство между ними? В каких случаях целесообразно применять BR, а в каких JMP? Возможно ли применение в программе только какой-либо одной из этих команд?

9. Что будет содержать ячейка A в конце выполнения следующей программы, записанной на MACRO-11?

```
        .=200
        R1=%1
        R2=%2
START:  MOV     #230,R1
        MOV     2(R1),R2
        MOV     R1,B
        ADD     B,R2
        MOV     R2,A
        HALT
B:      .WORD   0
C:      .WORD   230,20,54
A:      .WORD   0
        .END    START
```

10. Программа, приведенная в упр. 9, ассемблируется, начиная с ячейки 400. Конечное содержание ячейки A должно оставаться тем же самым, что и в упр. 9. Нужны ли какие-нибудь изменения? Объясните.

11. Сложите числа, находящиеся в ячейках 2004, 2006, ... и имеющие диапазон изменения от MIN до MAX. Значения MIN и MAX хранятся в ячейках 2000 и 2002 и могут принимать любые допустимые для PDP-11 значения (положительные или отрицательные). Значение N принадлежит интервалу $0 \leqslant N \leqslant 99$. Если $N = 0$, то результат должен быть равен нулю. Сумма чисел должна храниться в ячейке 1776.

12. Напишите программу для нахождения моды (наиболее часто встречающейся величины) множества из N чисел. Числа сохраняются будучи неупорядоченными, а равные хранятся в последовательных ячейках памяти. Покажите, что ваша программа будет работать при всех значениях N в диапазоне $0 < N < 99$.

13. Имея в виду общие принципы работы ассемблера, объясните, почему следующий сегмент программы будет ассемблирован так, как это показано. Произведите ассемблирование вручную. Какие нежелательные характеристики содержит этот код?

```
        .=1000
MOV:    MOV     SUB,ADD
SUB:    ADD     MOV,SUB
ADD:    SUB     ADD,MOV
```

14. Надо *увеличить* содержимое R4 на единицу и *уменьшить* содержимое R5 на два. Обычная последовательность команд для выполнения этих операций: INC R4 и SUB # 2,R5. Студент A утверждает, что одна команда CMPB (R4) +, @ — (R5) выполнит эту работу; студент B — что последовательность из двух команд TSTB (R4) + и TST —(R5) тоже годится. Сначала покажите, что способы, предложенные студентами, пригодны для решения задачи. Затем проведите *тщательный* анализ всех трех способов с целью определения их эффективности по *времени* выполнения и занимаемому *адресному пространству*. (Подсказка: проследите, как данные последовательности кодов обрабатываются различными фазами обработки команд (параграф 3.5.4), записывая «подходящие» относительные времена для выполнения каждой части каждой команды.) Какой способ наиболее подходит с точки зрения практики программирования?

БИБЛИОГРАФИЯ

Так как целью этой главы является введение в PDP-11, для дальнейшего чтения рекомендуются различные руководства фирмы Digital Equipment Corp. Сюда относятся «Processor Handbook» для PDP—11/04, 33, 45, 55, 60 и «Processor Handbook» для PDP-11/70. LSI-11 или PDP-11/03 описываются в «Microcomputer Handbook». В «PDP-11 Software Systems Handbook» описывается программное обеспечение фирмы DEC. И окончательная проверка того, как материал понят, — это составление и выполнение реальных программ для ЭВМ.

Естественно, что для этого нужна настоящая машина PDP-11.

Архитектура PDP-11 анализировалась в значительном числе исследований, краткое сообщение о которых дано Burr W. E. и Smith W. R. (1977) и более подробное — в специальном выпуске IEEE «Computer» (октябрь, 1977). Аспекты «стоимость/исполнение» разработки малых ЭВМ в основном на примере PDP-11/60, а также эволюция ЭВМ в целом освещены Mudge J. C. (1978). Strecker W. D. (1976) сделал сообщение по некоторым вопросам разработки кэш-памяти для семейства машин PDP-11.

Глава 4

СТРУКТУРЫ ПРОГРАММ И МЕТОДЫ ПРОГРАММИРОВАНИЯ

Знакомство с основами системы команд ЭВМ было только первым шагом в изучении программирования. Теперь на основании приведенных ниже приемов читателю предстоит научиться получать правильные результаты с помощью эффективных методов. Каждый рассматриваемый прием иллюстрирован примерами, которые представляют всю систему команд.

4.1. КОМАНДА ПЕРЕХОДА

Рассмотрим подробнее упомянутую ранее команду JMP. Необходимость в ней определяется относительной ограниченностью возможностей команд переходов, которые не могут передавать управление в точку, отстоящую от данной более чем на 128 слов в ту или другую сторону. Следовательно, для обеспечения перехода в произвольную область памяти необходима команда с адресным полем длиной в полное слово. Такая команда перехода JMP действительно имеется в группе одноадресных команд PDP-11. Она обеспечивает возможность передачи управления с использованием относительной, абсолютной, косвенной или индексной адресации. Гибкость в адресации может быть весьма полезной при программировании определенного класса задач. Продемонстрируем это на примере.

4.1.1. Задача о таблице переходов

Существует класс задач, в которых последовательность действий при решении находится в зависимости от входных данных. Для реализации этой зависимости может использоваться специальная переменная; на основе исходной информации ей присваивается определенное значение, которое в конечном итоге определяет выбор того или иного блока команд. На Фортране подобная задача может быть представлена так:

```
.
.
INDEX=..
GO TO (10,100,37,1150,...,7), INDEX
.
.
.
```

При выполнении этого фрагмента управление будет передаваться в соответствии со значением переменной INDEX: оператору с меткой 10 — при INDEX = 1, оператору с меткой 100 — при INDEX = 2, 37 — при INDEX = 3 и т. д.

Вычисляемый оператор GO TO должен быть переведен на машинный язык, и одно из возможных решений может быть таким:

```
        .
        .
        MOV     INDEX,R1        ;R1 ← INDEX
        DEC     R1              ;0 ≤ INDEX < max
        ADD     R1,R1           ;R1 ← 2*R1
        JMP     @TABLE(R1)      ;DEFFERED JUMP
        .
        .
TABLE:  .WORD L10,L100,L37,L1150,...,L7
        .
```

Этот прием программирования называют *методом таблиц переходов* в связи с использованием таблицы адресов переходов. Работает он следующим образом:

1. Из значения переменной INDEX, заключенного в пределах $1 \leqslant \text{INDEX} \leqslant \text{MAX}$, вычитается 1, для того чтобы пределы стали $0 \leqslant \text{INDEX} \leqslant \text{MAX}$.

2. Учитывая, что элементами таблицы являются 16-разрядные адреса, значение INDEX должно быть умножено на 2. Такого рода преобразование целого значения смещения в значение, выраженное в словах, нередко встречается в программах для PDP-11.

3. Исполнительный адрес команды перехода JMP выбирается из ячейки, заданной суммой (R1) + TABLE.

Команда перехода позволяет передать управление по заданной метке, но она не дает возможности задавать адрес возврата. В следующем разделе, рассматривая подпрограммы, мы покажем, что при небольшой модификации этой команды появляется возможность передавать управление из одной части программы в другую, а затем возвращаться обратно.

4.2. ПОДПРОГРАММЫ

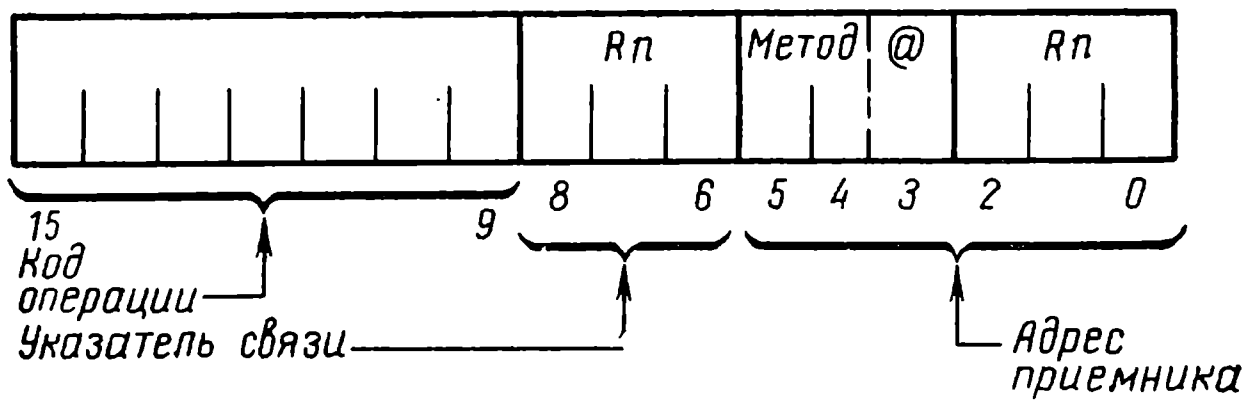
При программировании больших задач целесообразно разбивать всю программу на отдельные небольшие *подпрограммы,* которые можно независимо разрабатывать и отлаживать. При выполнении из главной программы осуществляются вызовы отдельных подпрограмм, каждая из которых может включать в себя вызов другой подпрограммы. После завершения выполнения управление возвращается в вызывающую программу, обычно на команду, следующую за вызовом. Распределение решения задачи между отдельными программными модулями позволяет более эффективно использовать память ЭВМ. Дополнительно появляется возможность избежать повторного кодирования тех из них, которые используются многократно. Применяя принцип модульности, можно строить целые библиотеки программ на языках высокого уровня или на языках ассемблеров отдельных машин. Это озна-

чает, например, что при решении сложных задач, *вычислительное ядро* которых имеет такие фундаментальные математические преобразования, как матричные операции, преобразование Фурье, автокорреляцию и т. п., можно воспользоваться готовыми стандартными программами.

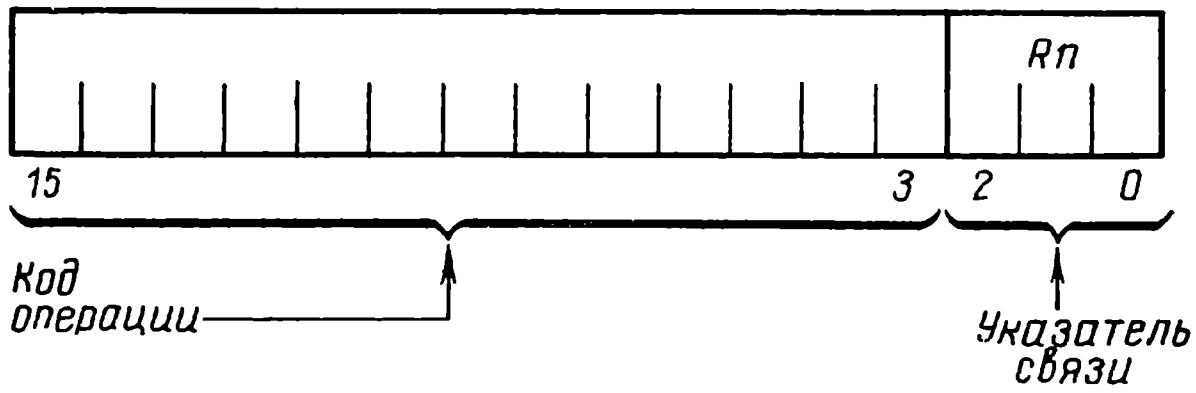
Для всего этого необходим некоторый механизм, который, во-первых, обеспечивает передачу управления от одной программы к другой и, во-вторых, реализует обмен данными между ними.

Механизм, удовлетворяющий этим двум требованиям и обычно называемый *аппаратом вызова подпрограмм,* в целом представляет собой сочетание аппаратных и программных средств. В системе PDP-11 средства вызова подпрограмм представлены двумя командами JSR и RTS, имеющими следующий синтаксис и формат:

JSR регистр, приемник



RTS регистр



Однако, прежде чем перейти к их детальному анализу, еще раз рассмотрим концепцию стека.

4.2.1. Стек

Стек — это область памяти, используемая для временного хранения данных. Он устроен по принципу «*последним пришел — первым ушел*»: записи извлекаются из стека в порядке, обратном их поступлению. В системе PDP-11 работа со стеками поддерживается наличием автоинкрементного и автодекрементного методов адресации, позволяющих динамически заносить, удалять, модифицировать отдельные записи стека или весь его целиком В PDP-11 стек в памяти начинается с ячейки с наибольшим из зарезервированных для него адресов и линейно распространяется по мере поступления новых записей в сторону уменьшения адресов.

При работе со стеком программист освобожден от необходимости следить за тем, в какие ячейки помещаются его данные. Эту функцию автоматически выполняет *указатель стека* — один из регистров общего назначения, всегда хранящий адрес последней произведенной в него записи. При организации стеков программист может использовать любые регистры в качестве указателей, за исключением регистра 7 (R7, PC). Аппаратом вызова подпрограмм (т. е. JSR и RTS) и системой обработки прерываний (см. разд. 7.5) в качестве *системного указателя стека автоматически используется* R6 (он обозначается SP).

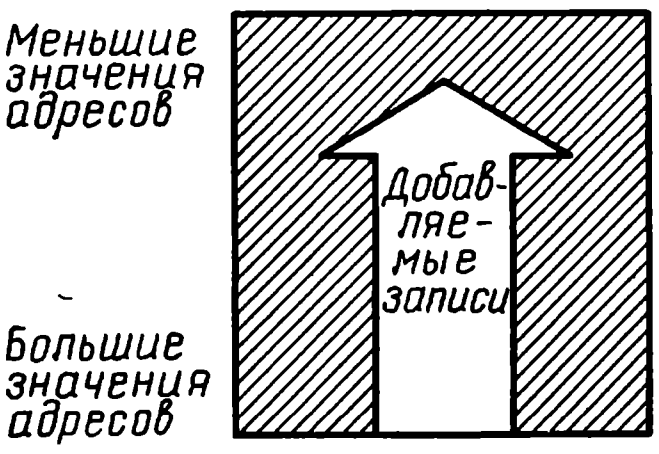


Рис. 4.1. Организация стека

Элементами стеков системы PDP-11 могут быть байты или слова. Исключение составляет системный стек, в котором хранятся только слова. Для занесения в стек используется автодекрементный метод адресации, например MOV ITEM, —(Rn), а для выборки — автоинкрементный метод, например MOV (Rn)+, ITEM. Примеры использования этих методов адресации делают понятным, почему при автодекрементном методе уменьшение содержимого регистра происходит до выполнения, а при автоинкрементном, наоборот, после выполнения команды. Дело в том, что указатель должен всегда показывать на последний элемент стека, поэтому его следует изменить до запоминания нового и соответственно после удаления существующего элемента из стека. Для побайтового чтения из стека по любому регистру, кроме R6, можно воспользоваться командой MOVB (Rn)+,ITEM, а для записи — MOVB ITEM, —(Rn).

Напомним, что указатель стека всегда показывает на последнюю использованную стеком ячейку и запись всегда выполняется в ячейку, соседнюю с последней и имеющую меньший адрес.

4.2.2. Подпрограммы: вызов и возврат

При выполнении команды перехода в подпрограмму в системном стеке по R6 сохраняется содержимое регистра возврата Rn так, как если бы была выполнена команда MOV Rn, —(SP). Затем в этот регистр загружается адрес, следующий за JSR в вызывающей программе (текущее состояние PC), а выполнение возобновляется с точки, указанной в команде вызова. В частности, при выполнении команды JSR Rn, DST адрес подпрограммы преобразуется следующим образом:

```
(TMP)  ←  (Rd)    ;SAVE Rd, (DST ADDRESS, SEC. 3.5.4b)
-(SP)  ←  (Rn)    ;PUSH Rn ONTO STACK
(Rn)   ←  (PC)    ;LOAD Rn WITH RETURN ADDRESS
(PC)   ←  (TMP)   ;TRANSFER CONTROL TO SUBROUTINE
```

Здесь TMP и Rd — внутренние регистры процессора. На рис. 4.2 показано состояние регистров и ячеек до и после выполнения JSR R5, SUBR; в этом примере первое слово команды JSR расположено в ячейке 1000, а значение символического адреса SUBR равно 1064.

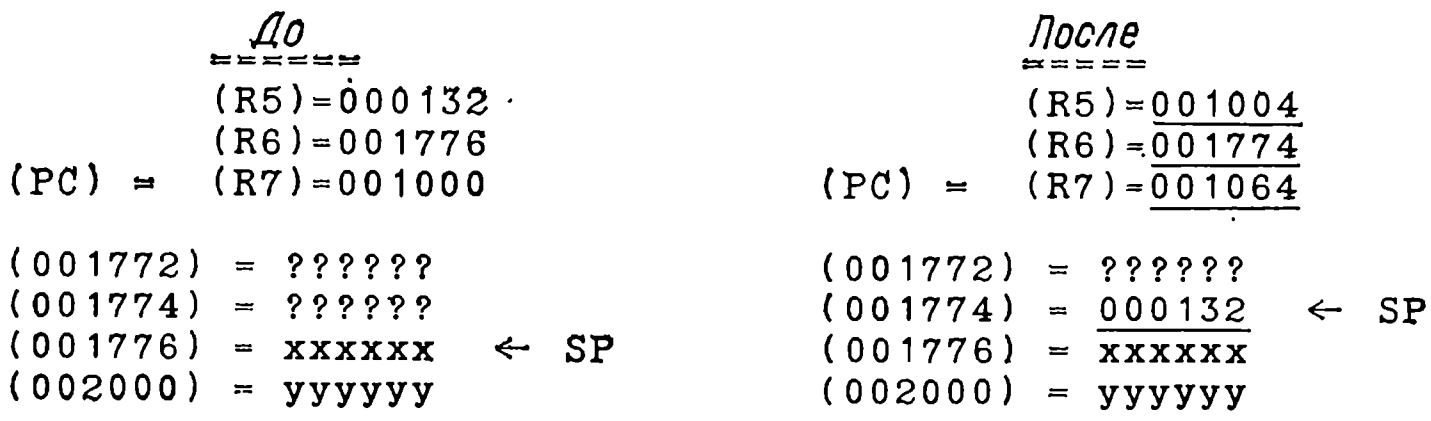


Рис. 4.2. Команда JSR

Команда RTS Rn используется для организации возврата из подпрограммы; ее действия обратны действиям JSR. RTS извлекает из стека и восстанавливает сохраненное значение регистра возврата Rn и передает управление в точку, следующую за JSR.

```
(PC) ← (Rn)     ;RESUME PROGRAM EXECUTION AT RETURN ADDRESS
(Rn) ← (SP)+    ;POP OLD Rn FROM STACK
```

Использование механизма стека для вызова и возврата из подпрограммы целесообразно по двум причинам. Во-первых, выполняя последовательность нескольких команд JSR, можно не заботиться о сохранении информации, требуемой для обратного перехода, поскольку все необходимое для этого записывается в стек в нужном порядке. Тогда достаточно просто может быть организован возврат — путем выполнения команды RTS с соответствующим регистром. При этом нужная информация автоматически выбирается из стека. Такой способ сохранения адресов связи автоматически обеспечивает *вложенность подпрограмм* и при необходимости допускает вызов подпрограммой самой себя (подробнее см. в параграфах 4.4.1 и 4.4.2). При анализе механизма прерываний (разд. 7.5) мы рассмотрим некоторые другие аспекты вызова подпрограмм.

Еще одно преимущество стекового механизма проявляется при сохранении и восстановлении регистров. В подпрограмме необходимость сохранения регистров возникает тогда, когда нужно использовать те из них, которые уже фигурировали в вызывающей программе. Содержимое этих регистров при входе в подпрограмму должно сохраняться, а при выходе — восстанавливаться. Возможные варианты оформления такой подпрограммы приведены на рис. 4.3.

На каждую из операций сохранения и восстановления регистров вторая программа требует вдвое меньше слов и позволяет

другим программам в дальнейшем использовать временно занятое ею пространство стека, тогда как первая исключительно для своих нужд резервирует несколько ячеек памяти (TEMP), которые в последующем не нужны. Организация временно используемых разделяемых областей в форме стека представляет собой

```
SUBR:   MOV     R0,TEMP          ;SAVE R0
        MOV     R1,TEMP+2        ;SAVE R1
        MOV     R2,TEMP+4        ;SAVE R2
        MOV     R3,TEMP+6        ;SAVE R3
        .
        .
        MOV     TEMP+6,R3        ;RESTORE R3
        MOV     TEMP+4,R2        ;RESTORE R2
        MOV     TEMP+2,R1        ;RESTORE R1
        MOV     TEMP,R0          ;RESTORE R0
        RTS     PC               ;RETURN
TEMP:   .WORD 0,0,0,0,0,0        ;SAVE AREA


SUBR:   MOV     R0,-(SP)         ;SAVE R0
        MOV     R1,-(SP)         ;SAVE R1
        .
        .
        MOV     (SP)+,R1         ;RESTORE R1
        MOV     (SP)+,R0         ;RESTORE R0
        RTS     PC
```

Рис. 4.3. Использование стека для сохранения и восстановления содержимого регистров

весьма экономичный способ резервирования памяти и имеет особо важное значение в тех случаях, когда ее доступный объем ограничен.

Очевидно, что для вызова подпрограммы вполне можно употребить команду JSR PC, SUBR. Она не используется и не помещает в стек никаких регистров, кроме PC, т. е. с ее помощью в стек помещается собственно адрес возврата, который считывается по команде RTS. Читателю рекомендуется проследить выполнение команды JSR PC, SUBR на основе приведенного выше примера. Отметим, что если JSR с применением регистра PC вполне корректна, то команда JSR SP, SUBR, где SP = R6, воспринимается как ошибка.

4.2.3. Передача аргументов

Команды JSR и RTS выполняют голько функцию передачи управления при решении задачи взаимодействия основной программы и подпрограммы. Кроме того, необходимо передать в подпрограмму значение аргументов и получить их обратно. Для решения этой части задачи нужно выбрать один из нескольких возможных способов.

Самый простой подход состоит в том, чтобы заранее зарезервировать нужное число ячеек, которыми можно будет в дальнейшем воспользоваться. Допустим, например, существует некоторая

подпрограмма MULT, умножающая два 16-битовых числа; она получает 32-битовый результат. Предполагается, что множимое и множитель находятся соответственно в ячейках ARG1 и ARG2 и после исполнения результат остается там же.

В данном случае передача аргументов, вызов подпрограммы и сохранение результата может иметь следующий вид:

```
MOV     X,ARG1          ;MULTIPLIER
MOV     Y,ARG2          ;MULTIPLICAND
JSR     PC,MULT         ;CALL MULTIPLY
MOV     ARG1,RSLT       ;SAVE HIGH ORDER WORD
MOV     ARG2,RSLT+2     ;SAVE LOW ORDER WORD
```

Если же мы применим для передачи аргументов регистры, то подпрограмма будет выглядеть так:

```
MOV     X,R1            ;MULTIPLIER
MOV     Y,R2            ;MULTIPLICAND
JSR     PC,MULT         ;CALL MULTIPLY
```

Хотя последний метод вполне приемлем, он имеет ограничение в виде числа общих регистров, доступных пользователю, т. е. не может быть передано более шести аргументов. Альтернативой ему может служить подход, при котором для передачи аргументов используются ячейки памяти, адресуемые в команде JSR регистром возврата. Поскольку этот регистр указывает на первое слово, следующее за командой JSR, его можно интерпретировать как указатель либо вектора аргументов, либо вектора адресов аргументов. В первом случае за командой JSR непосредственно в памяти следует *собственно аргументы* и вызов подпрограммы можно представить так:

```
JSR     R0,MULT         ;CALL MULTIPLY
.WORD   XVALUE,YVALUE   ;ARGUMENTS
```

Доступ к этим аргументам возможен при помощи автоинкрементного метода адресации:

```
MULT:   MOV     (R0)+,R1        ;GET MULTIPLIER
        MOV     (R0)+,R2        ;GET MULTIPLICAND
        .
        .
        RTS     R0
```

К моменту исполнения возврата значение R0 будет увеличено на 4, и тогда R0 будет содержать адрес следующей за JSR исполняемой команды.

Если за командой вызова следуют *адреса аргументов,* можно записать:

```
JSR     R0,MULT         ;CALL MULTIPLY
.WORD   XADDR,YADDR     ;ARGUMENT ADDRESSES
```

И тогда доступ к аргументам осуществляется при помощи косвенной адресации:

```
MULT:   MOV     @(R0)+,R1       ;FETCH MULTIPLIER
        MOV     @(R0)+,R2       ;FETCH MULTIPLICAND
        .
        .
        RTS     R0              ;RETURN
```

На передаче адреса только первого элемента списка аргументов основан еще один метод. Если этот адрес поместить в один из регистров общего назначения, то отпадает необходимость размещать список аргументов в той же области памяти, где расположен вызов подпрограммы, и подпрограмма может работать с данными, расположенными произвольно. В таком случае операции выполняются над данными, которые расположены в стеке или же на которые в стеке имеются ссылки.

```
Item #1 <- R1 (points to item #1)
Item #2
```

Рис. 4.4. Использование стека для хранения аргументов подпрограмм

Вызывающая программа:

```
MOV     #POINTER,R1     ;SET UP POINTER
JSR     PC,SUBR         ;CALL SUBROUTINE
```

Подпрограмма:

```
        ADD     (R1)+,(R1)      ;ADD ITEM #1 TO ITEM #2
        .                       ;PLACE RESULT IN ITEM #2
        .                       ;R1 POINTS TO ITEM #2 NOW
или
        ADD     (R1),2(R1)      ;SAME EFFECT AS ABOVE,
                                ;BUT R1 STILL POINTS TO ITEM #1
```

После рассмотрения стольких механизмов передачи управления, естественно, возникает вопрос: нужны ли все они при реальном программировании? На это всегда можно ответить положительно, поскольку любой из методов в определенном смысле имеет преимущество над остальными. Здесь можно ограничиться следующими характеристиками:

1) использование меньшего числа регистров для передачи аргументов;

2) большее число передаваемых параметров;

3) простота механизма вызова.

Последний пункт нуждается в дополнительных пояснениях. Поскольку, как и любые программы, подпрограммы могут быть написаны в *позиционно-независимой форме*, а также создаваться и ассемблироваться независимо от основной программы, возникает проблема, как передать адрес подпрограммы в команду JSR.

При анализе вопроса о передаче аргументов нами не был рассмотрен еще один момент. Если в первых, простейших приме-

рах в подпрограмму передавались сами значения, то в последующих — адреса значений. Различие между этими двумя способами — *вызовом по значению и вызовом по адресу* — может быть довольно серьезным; мы проиллюстрируем это на примере программы, написанной на языке, подобном Фортрану:

```
C       PROGRAM TRICKY,                 SUBROUTINE SWAP(X,Y)
        A=1.                            TEMP=X
        B=2.                            X=Y
        PRINT, A-B                      Y=TEMP
        CALL SWAP(1.,2.)                RETURN
        A=1.                            END
        B=2.
        PRINT, A-B
        STOP
        END
```

Если действительная константа передается по значению, то оба оператора печати напечатают —1. Это произойдет потому, что подпрограмма SWAP поменяет местами значения, которые она получит, а не собственные значения аргументов.

Однако, если действительная константа передается по адресу, то на печать будет выдано —1. и 1. соответственно. В этом случае подпрограмма SWAP обратилась бы к самим константам и поменяла их значения.

В таких языках высокого уровня, как Фортран, параметры могут передаваться и по значению, и по адресу. Нормальным обычно является режим передачи по адресу, но когда аргументом является выражение, то адресом является адрес ячейки, хранящей его вычисленное значение. Поэтому, если есть необходимость в вызове подпрограммы SWAP с передачей аргументов по значению, вызов следует оформить так:

```
CALL SWAP(1.*1.,2.*1.)
```

обеспечивая тем самым перестановку значений выражений, а не собственно аргументов.

Приведенные приемы передачи аргументов понятны на уровне языка Ассемблера, когда очевиден применяемый прием. При работе на языках высокого уровня это не всегда так, вследствии чего иногда получаются неожиданные результаты. Поэтому опытный программист, работающий на языке высокого уровня, должен учитывать особенности используемых методов, для того чтобы избежать неожиданностей.

4.2.4. Использование регистров в подпрограммах

Подпрограммы, как и программы, во время выполнения используют регистры. В итоге их содержимое до входа в подпрограмму и после выхода из нее различается. В связи с этим и требуется их распределение как системных ресурсов с реализацией сохранения и восстановления.

Ответственность за выполнение этой функции ложится либо на вызывающую программу, либо на вызываемую подпрограмму. Поскольку необходимо сохранить текущее состояние на момент вызова, то принципиально запоминание и восстановление допустимо в вызывающей программе, но чаще это делает сама подпрограмма. В PDP-11 использование стека сильно упрощает такую процедуру (см. рис. 4.3).

Стек, как указывалось выше, заполняется в сторону уменьшения адресов, обычно место для него резервируется непосредственно перед соответствующей программой или программами. Первое, что должна сделать программа,— это установить начальное состояние его указателя *. Например, если применяется SP (т. е. R6), то программа должна начинаться следующим образом:

```
BEG:    MOV     #BEG,SP         ;BEG IS THE FIRST INSTRUCTION
                                ;SP <- ADDR OF BEG
                                ;FIRST ITEM PUSHED ONTO STACK
                                ;WILL RESIDE AT ADDR BEG-2
```

К сожалению, такая программа не может быть размещена в произвольном месте в памяти, т. е. ее можно назвать позиционно-зависимой. Теперь рассмотрим программы, обладающие свойством позиционной независимости.

4.3. ПОЗИЦИОННО-НЕЗАВИСИМОЕ ПРОГРАММИРОВАНИЕ

Программы, создаваемые в рамках концепции модульного программирования, в соответствии со сказанным выше должны быть стандартны, доступны различным пользователям. Так как не известно заранее, куда будет загружена такая программа, требуется реализация возможности ее загрузки и выполнения в произвольном месте памяти. Для достижения этого существует несколько способов:

1. Повторно ассемблировать программу так, чтобы она начиналась с требуемой ячейки.
2. Использовать перемещающий загрузчик, который принимает от ассемблера специально подготовленные перемещаемые двоичные коды.
3. Создавать программы, перемещающие себя после загрузки.
4. Писать позиционно-независимые программы.

Повторное ассемблирование чаще используется на малых машинах. Если позволяют ресурсы памяти, то предпочтительнее иметь *перемещающий загрузчик.* Программы, перемещающие себя, оказываются экономически нецелесообразными, поскольку требуют перестройки сотен или даже тысяч адресов. Возможность создания позиционно-независимых программ определяется струк-

---

* Сказанное справедливо для программ, функционирующих не под управлением операционной системы, поскольку в противном случае указатель стека уже установлен. — *Примеч. ред.*

турой машины, на которой они реализуются. В частности, система PDP-11 допускает позиционно-независимое кодирование (PIC) программ.

Для этого служат те методы адресации, в которых исполнительный адрес формируется относительно счетчика команд (PC). Следовательно, при любом перемещении команды перемещаются и другие связанные с ней элементы программы, а расстояние между ними не меняется. Такое постоянное относительно PC значение смещения можно вычислять для всех адресов программы. При PIC для ссылки на ячейку используется ее смещение относительно той, в которой осуществляется обращение. Абсолютные ссылки в PIC-программах возможны только к тем ячейкам памяти, которые занимают постоянное место даже если сама программа перемещается.

4.3.1. Позиционно-независимые методы адресации

Существуют три типа позиционно-независимых команд.

1. *Переходы* — команды условного перехода и безусловный переход JMP позиционно-независимые, в них адрес вычисляется как смещение относительно счетчика команд PC.

2. *Относительные ссылки к ячейкам памяти* — ссылки в форме «CLR X» или «MOV X, Y» позиционно-независимые, потому что смещение в этих командах ассемблируется относительно значения PC. Под смещением понимается расстояние между текущим значением PC и ячейкой, на которую выполняется ссылка; например, если команда CLR X имеет адрес 100, а X отстоит от нее на 100 байт, т. е. имеет адрес 200, тогда:

```
Адрес   Содержимое   Символическое обозначение   Комментарии
------  ----------   -------------------------   -----------
000100  005067       CLR     X                   ;FIRST WORD
000102  000074                                   ;OFFSET=200-104
000104  ??????       next instruction
```

Смещение суммируется с содержимым PC. В этом примере значение PC равно адресу команды, следующей за ячейкой, содержащей смещение,— 104 (CLR X — двухсловная команда). Заметим, что в форме CLR X команда позиционно-независимая, а в форме CLR @X — позиционно-зависимая. В подтверждение этого рассмотрим следующий пример:

```
Адрес   Содержимое  Метка  Символическое обозначение   Комментарии
------  ----------  -----  -------------------------   -----------
001000  005077      S:     CLR     @X                  ;CLEAR LOCATION A
001002  000774             .
                           .
002000  003000      X:     .WORD   A                   ;POINTER TO A
                           .
                           .
003000  000001      A:     .WORD   1
```

Здесь содержимое ячейки X используется как адрес операнда, имеющего метку A, т. е. как абсолютный адрес ячейки A. При перемещении программы после ассемблирования ячейка A получит новый абсолютный адрес, и это должно быть отражено в значении ячейки X. Если же ячейка A имеет фиксированный абсолютный адрес, то операторы S и X могут быть перемещаемыми. В связи с этим в приведенном ниже примере программа будет позиционно-независимой.

```
                                Символическое
Адрес   Содержимое  Метка       обозначение        Комментарии
------  ---------  -----       -----------        ----------
        000036                  A=36
001000  005077     S:           CLR      @X        ;CLEAR LOCATION A
        000774
                                .
                                .
002000  000036     X:           .WORD    A         ;POINTER TO A
```

3. *Непосредственные операнды* — адресная форма Ассемблера служит для непосредственного задания данных, в этом случае часть команды ассемблируется как значение операнда.

Непосредственные операнды могут быть позиционно-независимыми только в том случае, если ими представлены константы, а не адреса. Тогда команда SUB #2, HERE позиционно-независимая, так как здесь #2 не адрес; в то же время команда MOV #A, ADRPTR позиционно-зависимая, поскольку A — символический адрес. Это справедливо, несмотря на то, что в обоих случаях операнд выбирается с использованием PC в автоинкрементном режиме.

4.3.2. Абсолютные методы адресации

Абсолютной, в противоположность относительной, называется такая ссылка, в которой в качестве указателя используется ячейка или регистр. Абсолютные методы адресации следует применять, если данные всегда, вне зависимости от положения PIC-программы, занимают фиксированную позицию. И наоборот, если данные занимают относительное положение к коду программы, то абсолютными методами воспользоваться нельзя, хотя и имеется возможность модификации указателей. Другими словами, если адресация осуществляется с помощью прямого и относительного режимов, то она позиционно-независимая, а при косвенном и либо абсолютном, либо относительном режимах — позиционно-зависимая. Например, команда

```
MOV @#X;HERE
```

т. е. «переслать содержимое слова, указанное PC, в данном случае это ячейка X, в слово, заданное относительно PC и имеющее символическое имя HERE». Пока позиция X остается в памяти неизменной, эта команда, имеющая один косвенный и один относительный операнд, может перемещаться в памяти.

Следующие методы адресации абсолютны:

```
@X                  Location X is a pointer
@#X                 The "immediate" word is a pointer
(R)                 The register is a pointer
(R)+ and -(R)       The register is a pointer
@(R)+,@-(R)         The register points to a pointer
X(R)                The base, X, plus (R) is the address of the operand
@X(R)               The base plus (R) is a pointer
```

Некоторого пояснения требует применение индексного метода. Как было показано в гл. 3, форма X(PC) — это обычный относительный метод доступа в память. В общем случае такой метод, использующий регистр, рассматривается как абсолютный, однако его можно употреблять и как относительный и, следовательно, использовать в PIC-программах. Индексный метод применяется для доступа к данным, расположенным относительно указателя, адресующего некоторую динамическую область памяти. Если задать ему начальное значение, то ссылки становятся относительными.

4.3.3. Как писать PIC-программы, не требующие дополнительной настройки

Не требующими настройки называют такие PIC-программы, в которых нет необходимости модифицировать адреса и указатели. Это значит, что ссылки ограничены относительными методами адресации даже если соответствующие ячейки фиксированы в памяти. Кроме того, нужно руководствоваться следующими рекомендациями:

1. Начинать программу со стартового адреса 0 для более удобного использования абсолютного загрузчика.

2. Все установки начальных значений должны иметь форму: «.=.+X» или «.=.—X» или «.= функция или символ, определенные внутри PIC-программы». Например, .=A+10, где A — локальная метка.

3. В программе не должны устанавливаться начальные значения адресов, т. е. при загрузке PIC-программы нельзя инициализировать начальные значения такими операторами, как

```
.=177404          ;SET LC TO 177404
.WORD 5,-512.     ;PRE-LOAD 177404, 177406
```

При загрузке PIC-программы абсолютным загрузчиком в память все данные размещаются с соответствующим загрузочным смещением (см. гл. 8). Те из них, которые должны размещаться в ячейках с абсолютными адресами, следует туда переслать во время выполнения, например:

```
MOV     #5,@#177404       ;LOC 177404 ← 5
MOV     #-512.,@#177406   ;LOC 177406 ← -512.
```

4.3.4. Написание настраиваемых PIC-программ

Часто бывает так, что написать PIC-программу без дополнительной настройки не удается; в таких случаях можно использовать некоторые методы, суть которых заключается в том, что часть перемещений выполняется в процессе работы программы.

Ниже изложена часть употребляемых для этой цели приемов программирования. В основном в них для определения места, где расположена программа, используется анализ счетчика PC. Зная положение программы, довольно просто определить параметр перемещения. Во всех приведенных ниже примерах предполагается, что программа была ассемблирована с начального адреса 0, а затем загружена абсолютным загрузчиком.

Инициализация фиксированных ячеек. Рассмотрим приведенный выше пример обнуления ячейки A с использованием косвенной адресации. Указатель на это слово, задаваемый как X, должен изменяться при перемещении программы. В сегменте программы, приведенном на рис. 4.5, его значение вычисляется при каждом исполнении. В данном примере оно не зависит от значения счетчика ячеек во время ассемблирования и определяется только значением счетчика PC во время загрузки программы.

```
000000 010700   S:      MOV     PC,R0           ;R0 ← ADDR OF S+2
000002 062700           ADD     #A-S-2,R0       ;ADD IN OFFSET
       001776
000006 010067           MOV     R0,X            ;MOVE POINTER TO X
       000766
000012 005077           CLR     @X              ;CLEAR VALUE
       000762                                   ;    INDIRECTLY
000016 000000           HALT
                ;
                ;
                        .=.+760
001000 002000   X:      .WORD   A               ;POINTER TO A
                ;
                ;
                        .=.+776
002000 000001   A:      .WORD   1               ;VALUE TO BE CLEARED
                        .END
```

Рис. 4.5. PIC-программа, требующая дополнительной настройки

Если теперь эту программу загрузить не с адреса 2000, а с какого-либо другого, то в ней не изменится ни один параметр, в листинге (см. рис. 4.5) изменится только левая колонка адресов.

Такой пример демонстрирует использование смещения $A - S - 2 = A - (S + 2)$, где $S + 2$ — значение счетчика PC, пересылаемое в R0 по команде MOV PC, R0. Смещение не зависит от начального адреса загрузки, во время ассемблирования он предполагается равным 0 и поэтому начальное значение смещения вычислится из $A - PC_{нулевое}$, где $PC_{нулевое} = S + 2$.

Позднее, при перемещении программы в процессе ее выполнения, в R0 уже будет храниться не исходное $PC = PC_{нулевое}$, а текущее состояние $PC = PC_{новое}$. Однако команда сложения

ADD продолжает суммировать с R0 непосредственное значение A—$PC_{нулевое}$, получая при этом

$$PCn+(A-PCz) = A+(PCn-PCz)$$

Это значение и является желаемым результатом, поскольку оно дает новый абсолютный адрес ячейки A, т. е. адрес A при ассемблировании плюс фактор смещения $PC_{новое} - PC_{нулевое}$.

Перемещение указателей. Если применение указателей необходимо, то, как только что было показано, их можно перемещать. Допустим, например, что для доступа к списку данных используется команда

```
ADD     (R0)+,R1
```

Указатель списка L может быть вычислен во время исполнения следующим образом:

```
M:      MOV     PC,R0           ;GET CURRENT PC
        ADD     #L-M-2,R0       ;ADD OFFSET
```

Если в программе определены несколько указателей, то их можно свести в одну таблицу; тогда вычисленный один раз фактор смещения может быть в цикле применен ко всем указателям. На рис. 4.6 приведен пример программы, иллюстрирующей этот прием. При выполнении упр. 3 этой главы читатель должен проверить справедливость следующего: при размещении программы с адреса 10000 в таблице PTRTBL значения указателей соответственно будут 10000, 10020 или 10030.

```
000000 010700   X:      MOV     PC,R0           ;PC POINTS TO X+2
000002 062700           ADD     #X-X-2,R0       ;R0 <- RELOCATION FACTOR
       177776
;
000006 012701           MOV     #PTRTBL,R1      ;GET A POINTER
       000030
000012 060001           ADD     R0,R1           ; AND RELOCATE IT
;
000014 012702           MOV     #TBLLEN,R2      ;GET TABLE LENGTH
       000003
;
000020 060021   LOOP:   ADD     R0,(R1)+        ;RELOCATE AN ENTRY
000022 005302           DEC     R2              ;COUNT DOWN
000024 001375           BNE     LOOP            ;STOP WHEN DONE
000026 000000           HALT
       000003           TBLLEN=3
000030 000000   PTRTBL: .WORD   X,LOOP,PTRTBL
000032 000020
000034 000030
       000001           .END
```

Рис. 4.6. Указатели перемещения

Нужно быть внимательными при повторном выполнении программы, содержащей таблицу перемещаемых указателей. Такое перемещение должно осуществляться строго один раз после загрузки программы, поэтому при повторном исполнении сегмент программы, выполняющий перемещение указателей, необходимо обойти.

4.4. РАЗВИТИЕ КОНЦЕПЦИИ ПОДПРОГРАММ

Особенно явно преимущества стековой организации проявляются при работе в сложных ситуациях, возникающих в программных системах, где на ограниченные ресурсы претендуют несколько конкурирующих задач. Такие ситуации могут встречаться в различных ЭВМ, начиная от небольших однопользовательских систем (при необходимости совместить выполнение фоновой задачи с вводом-выводом) до сложных мультипрограммных систем, управляющих комплексом пользовательских и системных программ. В любом случае от программного обеспечения требуется гибкость, экономия времени выполнения и используемой памяти.

4.4.1. Реентерабельность

Реентерабельностью называют свойство программы, позволяющее нескольким задачам одновременно пользоваться одной ее копией, размещенной в памяти. От обычной программы реентерабельная отличается тем, что очередная задача может воспользоваться ею, не ожидая завершения работы программы в другой задаче, иначе говоря, в одной и той же реентерабельной программе могут находиться несколько задач в разной степени завершенности. На рис. 4.7 демонстрируется использование реентерабельной программы.

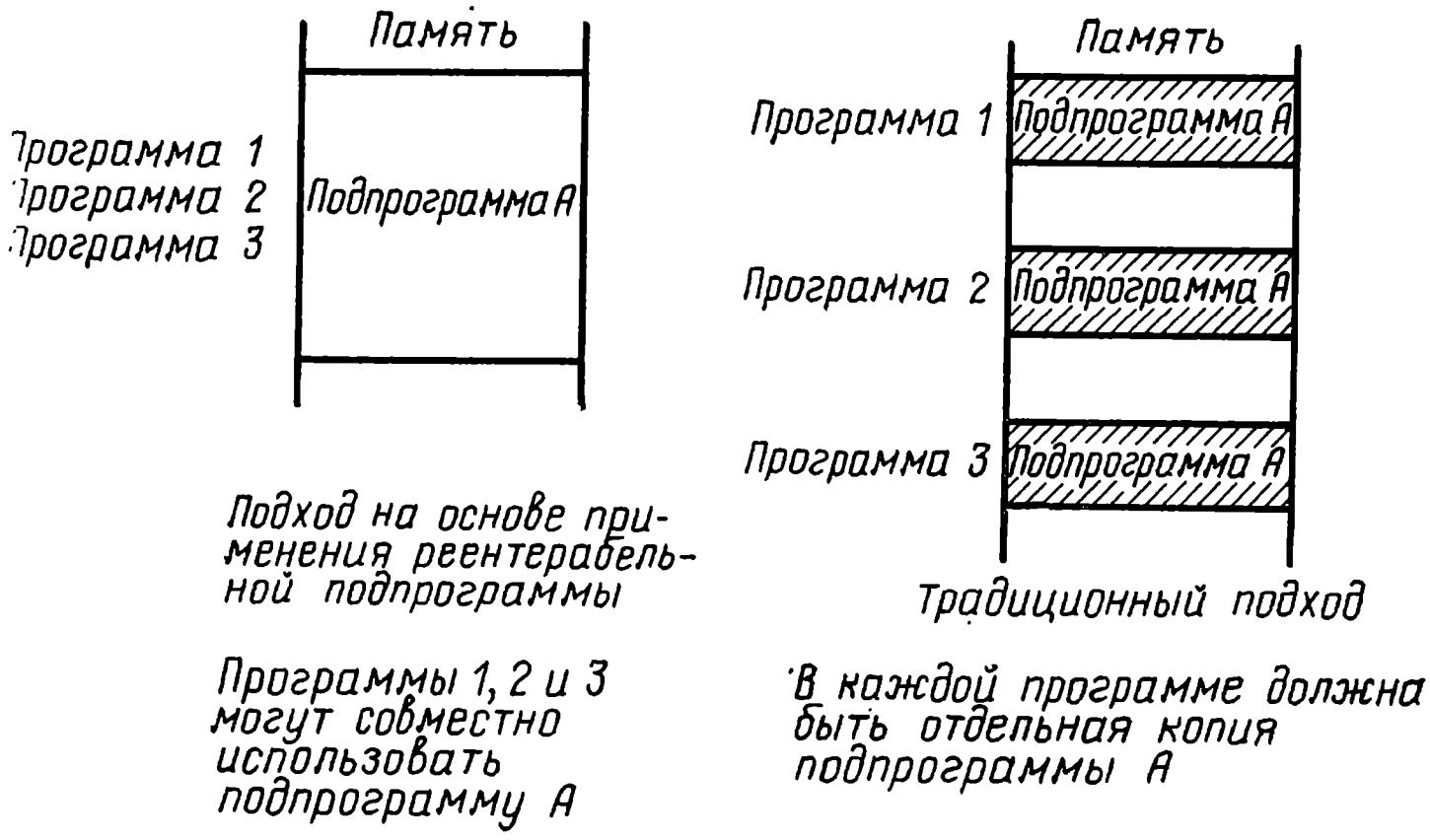


Рис. 4.7. Реентерабельные программы

Она характеризуется тем, что не содержит ничего, кроме *чистых кодов,* т. е. кодов команд и констант. В такой программе не должны использоваться области собственной памяти для записи или чтения, т. е. она может быть реентерабельной в том и только том случае, если не модифицирует свои коды во время выполнения. В целом понятие немодифицируемости программы шире реентерабельности. При отсутствии собственных областей

памяти для временного сохранения программы характеризуются следующими качествами:

1) простота отладки;

2) защита от записи, такие программы можно загрузить в постоянную память;

3) допустимость прерывания программы, а затем продолжения ее выполнения.

На рис. 4.8 показано использование двумя задачами одной реентерабельной программы.

Процедура взаимодействия в этом случае может быть описана так:

1. Задача А запросила работу реентерабельной программы *Q* как некоторого собственного компонента.

2. Выполнение других программ задачи А прерывается и Q начинает обрабатывать полученный запрос.

3. Обладающая более высоким приоритетом задача В также запросила работу программы Q.

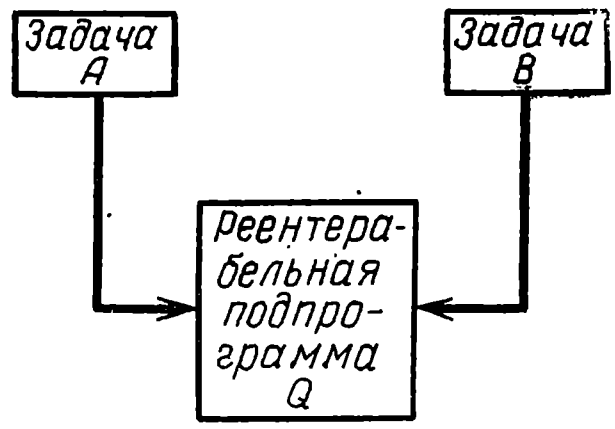


Рис. 4.8. Использование реентерабельной программы в двух задачах

4. Программа Q прерывает выполнение запроса от задачи А и приступает к обработке вновь поступившей. При этом она рассматривается уже как часть задачи В. По окончании работы Q продолжают выполнение соответствующие элементы задачи В.

5. Программа Q возобновляет обработку запроса задачи А и после ее завершения возвращает управление в последнюю. Задача А продолжает выполнение с точки, следующей за обращением.

Применение реентерабельных программ позволяет выделять функции, часто используемые различными задачами. Так могут быть оформлены программы перевода из кода в ASCII в двоичный, обслуживания устройств и др. В многопользовательской системе желательно сделать реентерабельный компилятор с Фортрана, тогда разные пользователи могли бы работать с одной и той же его копией.

4.4.2. Рекурсивность

Бывают случаи, когда целесообразен вызов программой самой себя; возможность организации его называют *рекурсивностью* программы. Математики часто пользуются рекурсией, во многих случаях она позволяет создавать простые алгоритмы для решения сложных задач. Мы покажем, что применение рекурсии улучшает использование памяти и скорость исполнения таких процедур, как лингвистические операции в компиляторах и других программах, работающих с языками высокого уровня.

Классическим примером рекурсии является вычисление факториала *N* (*N*!). Хотя справедлива запись

```
N! = N * (N-1) * (N-2) * .... * 1
```

но можно записать и так:

$$N! = N * (N-1)!$$

и

$$0! = 1.$$

На «псевдо-Фортране» вычисление функции *N*! можно записать таким образом:

```
      INTEGER FUNCTION FACT(N)
      IF (N.NE.0) GO TO 1
      FACT = 1
      RETURN
1     FACT = N * FACT(N-1)
      RETURN
      END
```

Большинство существующих реализаций компиляторов с Фортрана не могли бы сгенерировать правильный объектный код такой программы, поскольку реализация рекурсивного вызова требует обязательного использования стека для хранения текущих значений FACT и адресов возврата либо в саму подпрограмму-функцию, либо в вызывающую программу. В связи с тем, что такая возможность существует далеко не во всех компиляторах, мы и пользовались «псевдо-Фортраном». Однако на языке Ассемблера PDP-11 эта функция может быть запрограммирована с использованием стекового механизма системы. Если допустить, что значения *N* и значения *N*! записываются в стек, то функция FACTOR рекурсивно может быть закодирована следующим образом:

```
FACT:   MOV     N,-(SP)         ;INITIAL VALUE ON STACK
        JSR     PC,FACTOR       ;CALCULATE N!
        MOV     (SP)+,NFACT     ;GET ANSWER
        RTS     PC              ;RETURN TO CALLING PROGRAM
        ;
FACTOR: MOV     2(SP),R1        ;R1 <- N. ("2" INDEXES AROUND
                                ;         RETURN ADDRESS ON STACK)
        BEQ     ZERO            ;N = 0?
        DEC     R1              ;NO. R1 <- N-1
        MOV     R1,-(SP)        ;PUSH R1 = N-1 ONTO STACK
        JSR     PC,FACTOR       ;RECURSIVE CALL
RET:    MOV     (SP)+,R1        ;POP (N-1)! INTO R1
        MUL     2(SP),R1        ;R1 <- (N-1)!*N
        MOV     R1,2(SP)        ;N! = (N-1)!*N REPLACES N
        RTS     PC
ZERO:   MOV     #1,2(SP)        ;0! = 1
        RTS     PC
```

Рис. 4.9. Рекурсивная программа для вычисления факториала

Представленная на рис. 4.9 программа рекурсивно вызывает сама себя при выполнении команды JSR PC, FACTOR. Делая это, она каждый раз записывает в стек уменьшенное значение *N* и адрес возврата, имеющий метку RET.

Команда MUL умножает содержимое R1 на число, которое представляет собой промежуточный результат (изначально 1), хранимый в стеке.

После умножения выполняется команда возврата RTS. В этот момент в стеке может быть либо адрес возврата в программу, вызвавшую FACTOR, либо очередная двухсловная запись, выполненная при рекурсивном вызове подпрограммы FACTOR. В последнем случае в R1 загружается новое значение *N*, которое будет умножено на промежуточный результат, хранимый в стеке, и процесс повторится. Если же вычисление факториала закончено, то управление передается в вызывающую программу, а результат остается в стеке.

4.4.3. Сопрограммы

Встречаются ситуации, когда несколько программ должны находиться в режиме активного взаимодействия, при этом управление неоднократно передается от одной из них к другой. Каждая такая программа периодически ожидает передачи ей управления. Отношения между этими программами симметричны, поэтому их называют *сопрограммами.*

Идея сопрограмм в основном является развитием концепции подпрограмм. Но в отличие от подпрограммы сопрограмма не находится в подчиненном отношении по отношению к вызывающей программе. Соответственно подпрограммы и сопрограммы различаются по способам передачи управления.

При вызове подпрограммы основная программа прерывает свое выполнение. Работа подпрограммы начинается с самого начала, и после завершения она возвращает управление в вызывающую программу.

При передаче управления в подпрограмму выполнение возобновляется с того места, где она была прервана, не обязательно с начала. При функционировании сопрограмм управление последовательно передается от одной к другой, и та из них, которая получает управление, производит некоторое действие, продвигающее вперед весь процесс.

Использование аппаратных возможностей стека PDP-11 позволяет установить между двумя сильно связанными процедурами отношения, при которых они взаимодействуют как сопрограммы. Это достигается с помощью специального режима команды JSR [JSR PC, @(R6)+], который позволяет выполнить обмен содержимым между верхним элементом стека, адресуемого регистром R6, и счетчиком команд PC. Тогда программы могут передавать управление друг другу, возобновляя выполнение с того места, где оно было прервано. Рис. 4.10 иллюстрирует такое взаимодействие. Заметим, что в этом случае стек должен быть инициализирован командой MOV #PC2, —(R6).

Хотя аппарат сопрограмм находит широкое применение при создании современных операционных систем, но в целом этот вопрос выходит за рамки нашего изложения. В гл. 7 мы продемонстрируем использование сопрограмм для двойной буферизации ввода-вывода, совмещаемого с вычислением. Несмотря на некоторую

простоту, этот пример демонстрирует одну из основных операций ввода-вывода в большинстве операционных систем.

Работает программа # 1, она выполняет команду

JSR PC, @(R6)+

при этом происходит следующее:
1) PC2 выбирается из стека и SP получает автоинкрементное приращение;
2) SP автоинкрементно уменьшается и старое значение PC, т. е. PC1, помещается в стек:
3) управление передается в ячейку PC2, т. е. в программу # 2.

Работает программа # 2, где команда

JSR PC, @(R6)+

меняет местами в стеке PC2 и PC1, и управление снова передается в программу # 1.

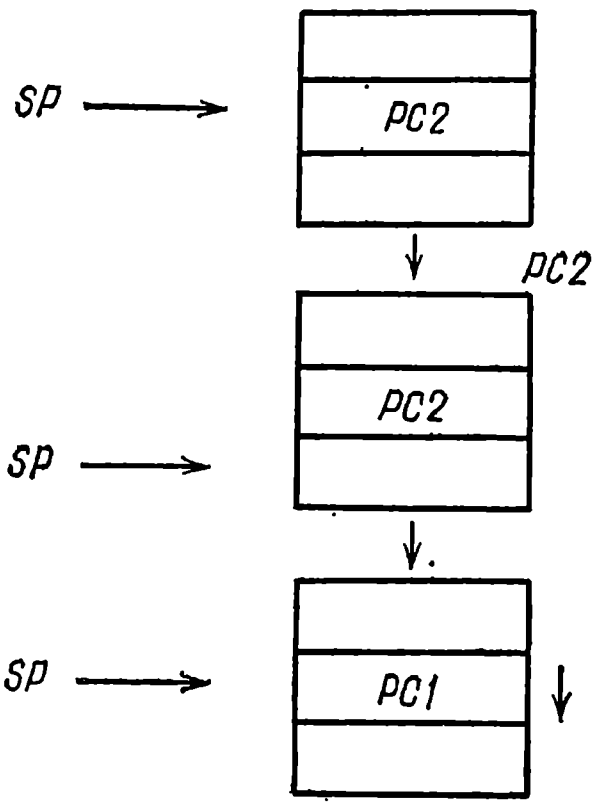


Рис. 4.10. Взаимодействие сопрограмм

Читателю рекомендуется, воспользовавшись протоколом из разд. 4.4.2, проследить выполнение команды JSR PC, @(R6)+.

4.5. МАКРОКОМАНДЫ И МАКРОАССЕМБЛЕРЫ

Использование подпрограмм не является единственным приемом для повторения *одних и тех же* последовательностей в *различных* местах программы. Аналогичным свойством обладают *макрокоманды*, но, прежде чем рассматривать их, обсудим основные принципы организации *макроассемблера*.

Читателю уже известно, что применение языка символического Ассемблера упрощает программирование, известно и назначение директив, предписывающих ассемблеру определенные действия. Теперь мы рассмотрим еще одно средство ассемблера, называемое *макрогенератором* и представляющее собой расширение языка. Обратите внимание на слово «расширение»: макроассемблер как средство программирования состоит из обычного символического Ассемблера и средств для обработки макрокоманд.

Одно из достоинств генератора макрокоманд состоит в том, что он позволяет работать с повторяющимися последовательностями команд, с помощью простых средств передачи параметров. Макрогенератор также позволяет программисту создавать новые элементы языка, приспосабливая Ассемблер для собственных нужд. При этом допускается многоуровневая вложенность макрокоманд, а также переопределение макрокоманд в программе.

Теперь перейдем от определения того, что можно получить от макрокоманды, к описанию того, как она работает. По существу, макрокоманда представляет собой *открытую подпрограмму,* описанную в виде формальной последовательности команд и заменяемую при вызове их фактической последовательностью. Основ-

ное преимущество применения макрокоманд заключается в повышении удобства программирования. хотя в некоторых случаях кроме этого удается экономить память или повышать быстродействие программы.

В программах достаточно часто встречаются повторяющиеся последовательности команд, которые отличаются только значениями аргументов. В таком случае, безусловно, удобно заменить всю последовательность одним оператором. Для того чтобы сделать это, нужно сначала определить соответствующую макрокоманду, а затем, при вызове, использовать ее имя, но уже со списком фактических параметров, которые при генерации заменяют собою формальные. В результате будет получена желаемая последовательность команд.

Итак, прежде всего макрокоманду необходимо определить. Ее определение должно начинаться директивой .MACRO и заканчиваться директивой .ENDM, например:

```
.MACRO  MAC                    ;MACRO NAME
LINE1
LINE2                          ;MACRO BODY
LINE3
.ENDM                          ;TERMINATE MACRO
```

При вызове имя макрокоманды заменяется макрорасширением

Вход макропроцессора		*Выход макропроцессора*	
ADD	A,B	ADD	A,B
MOV	B,A	MOV	B,A
MAC		LINE1	
		LINE2	
		LINE3	
MOV	C,B	MOV	C,B

Этот процесс замещения выполняется до ассемблирования и представляет, по существу, просто символьную построчную подстановку.

Для того чтобы можно было использовать макрокоманду с различными данными, их необходимо включить в макровызов в качестве аргументов. Например, макрокоманда сложения двух байт ADDB может быть представлена так:

```
.MACRO ADDB X,Y
MOV     R0,-(SP)    ;SAVE R0
MOV     R1,-(SP)    ;SAVE R1
MOVB    X,R0        ;R0 <- BYTE 1, SIGN EXT. TO HIGH BYTE
MOVB    Y,R1        ;R1 ← BYTE 2, SIGN EXT. TO HIGH BYTE
ADD     R0,R1       ;RESULT
MOVB    R1,Y        ;RESULT INTO Y
MOV     (SP)+,R1    ;RESTORE R1
MOV     (SP)+,R0    ;RESTORE R0
.ENDM
```

Заметим, что при выполнении команд ее макрорасширения значение кодов условий не совпадет с соответствующими кодами команды ADD даже при отсутствии ошибок.

4.5.1. Генерация локальных переменных

При разработке макрокоманд возникает определенная проблема с метками внутри макроопределений. К примеру, *перепись блока данных* могла бы осуществляться следующим образом:

```
        .MACRO BLKMOV X,Y,N
        MOV     #X,R0           ;X IS STARTING ADDRESS OF ARRAY 1
        MOV     #Y,R1           ;Y IS STARTING ADDRESS OF ARRAY 2
        MOV     N,R2            ;NUMBER OF POINTS IN EACH ARRAY
LOOP:
        MOV     (R0)+,(R1)+     ;MOVE DATA
        SOB     R2,LOOP
        .ENDM
```

К сожалению, такая запись неверна, потому что при каждом вызове макрокоманды BLKMOV метке LOOP присваивалось бы новое значение, что регистрируется как ошибка.

У этой проблемы имеются два решения.

1. Метку можно сделать параметром макрокоманды, например:

```
        .MACRO BLKMOV X,Y,N,L
        .
L:
        .
```

2. Предоставить программисту возможность сообщать ассемблеру, что определенные имена являются внутренними идентификаторами макрорасширения и что ассемблер должен заменить их созданными им самим новыми, уникальными для каждого макровызова. Например, при исполнении макрокоманды

```
        .MACRO BLKMOV X,Y,N,?L
        .
L:
        .
```

будут сгенерированы следующие коды:

```
BLKMOV A,B,NPTS         MOV     #A,R0
                        MOV     #B,R1
                        MOV     NPTS,R2
                64$:    MOV     (R0)+,(R1)
                        SOB     R2,64$

BLKMOV C,D,NPTS         MOV     #C,R0
                        MOV     #D,R1
                        MOV     NPTS,R2
                65$:    MOV     (R0)+,(R1)+
                        SOB     R2,65$
```

Порожденные ассемблером идентификаторы меток всегда локальны, они представляют собой число в пределах 64 ÷ 127 и следующий за ним служебный символ. Локальные идентификаторы меток генерируются макроассемблером в порядке возрастания в том случае, когда не происходит замещения формального параметра фактическим; если же в макровызове имеется соответствующий фактический параметр, то происходит обычная подстановка и генерация нового идентификатора не выполняется.

4.5.2. Вложенность макрокоманд

Макрокоманды могут быть вложены одна в другую, т. е. допускается вложенность макроопределений. Введем понятие уровня вложенности макроопределения. Будем считать *макроопределением первого уровня* самую внешнюю макрокоманду, имеющую непосредственное определение. Макрокоманды, определенные в макроопределении первого уровня, будем называть *макрокомандами второго уровня*, и т. д. Например:

```
.MACRO LEVEL1 A,B        ;START LEVEL 1 DEFINITION
ADD     A,B
.MACRO LEVEL2 C,D        ;START LEVEL 2 DEFINITION
SUB     C,D
.MACRO LEVEL3 E,F        ;START LEVEL 3 DEFINITION
ADD     E,F
ADD     F,F
.ENDM                    ;END LEVEL 3 DEFINITION
CLR     C
.ENDM                    ;END LEVEL 2 DEFINITION
CLR     A
.ENDM                    ;END LEVEL 1 DEFINITION
```

В момент начала ассемблирования определены только макрокоманды первого уровня, и, следовательно, только к ним можно обращаться обычным способом; в этот момент еще не раскрыты макроопределения второго и последующих уровней. В дальнейшем появляется возможность вызова макрокоманд любого уровня, естественно, при соблюдении соответствующей очередности. Приведенное выше макроопределение может быть использовано следующим образом:

	Вызов	*Расширение*		*Комментарии*
LEVEL1	X,Y	ADD	X,Y	Causes LEVEL2 to
		CLR	X	be defined.
LEVEL2	I,J	SUB	I,J	Causes LEVEL3 to
		CLR	I	be defined.
LEVEL3	Y,I	ADD	Y,I	
		ADD	I,I	

Если вызов LEVEL3 будет выполнен раньше, чем LEVEL 2, то это приведет к ошибке, потому что к нужному моменту еще не было обработано его макроопределение.

4.5.3. Макровызовы внутри макроопределений

Можно писать макровызовы в теле макроопределения, причем вызываемая макрокоманда не обязательно должна быть к этому моменту определена. Вложенные в макроопределение вызовы макрокоманд необходимо определить к моменту использования внешней макрокоманды.

Для примера рассмотрим макрокоманду SWITCH, которая передает содержимое буфера A в буфер B или из B в A:

```
.MACRO SWITCH A,B,TEMP,N
SAV012
COPY    A,TEMP,N
COPY    B,A,N
COPY    TEMP,B,N
UNS012
.ENDM
```

В качестве упражнения читателю рекомендуется модифицировать предыдущий пример:

```
        .MACRO COPY FROM,TO,COUNT,?L
        MOV     #FROM,R1
        MOV     #TO,R2
        MOV     COUNT,R0
L:      MOV     (R1)+,(R2)+
        SOB     R0,L
        .ENDM

        .MACRO  SAV012
        MOV     R0,-(SP)
        MOV     R1,-(SP)
        MOV     R2,-(SP)
        .ENDM

        .MACRO  UNS012
        MOV     (SP)+,R2
        MOV     (SP)+,R1
        MOV     (SP)+,R0
        .ENDM
```

4.5.4. Рекурсивные вызовы макрокоманд и условное ассемблирование

В макроопределении вполне допустимо использование вызова самой определяемой макрокоманды, нужно только задать условие, определяющее глубину вложения, т. е. момент завершения процесса макрорасширения. Следовательно, ассемблер должен содержать некоторые средства для задания такого рода условий. Этой цели вполне удовлетворяют методы условного ассемблирования, применение которых не ограничивается только рекурсивными макроопределениями.

Обычно директивы условного ассемблирования служат для выборочной трансляции исходного текста в зависимости от внешних условий. Они имеют следующий формат:

```
.IF  cond,argument(s)
```

где «cond» — условие, которому может соответствовать или не соответствовать аргумент. При этом выполняются следующие проверки:

1) соответствует ли значению выражение в поле аргумента;
2) определен или не определен аргумент;
3) определен ли идентификатор или адресное выражение;
4) соответствует ли значение строки символов.

Если условие выполняется, то ассемблируются строки, начиная со следующей за директивой условия и вплоть до оператора .ENDC, обозначающего конец действия условия. В противном случае эти строки пропускаются ассемблером.

Возможна вложенность условных директив. При этом каждому оператору условия .IF должен соответствовать закрывающий оператор .ENDC. При невыполнении самого внешнего уровня не ассемблируются все входящие в него предложения. Если же оно выполняется, то происходит ассемблирование до первого вложенного оператора .IF, а последующие предложения ассемблируются в зависимости от нового условия. Набор операторов .IF, вложенных один в другой, можно интерпретировать как схему совпадения «И». Например, если уровень вложенности достигает трех, то предложения, следующие за последним условным оператором, ассемблируются в случае выполнения всех трех условий.

Таблица 4.1. **Директивы условного ассемблирования**

Тип	Директива	Условие
Сравнение	.IF EQ	Аргумент = 0
	.IF NE	Аргумент ≠ 0
	.IF GT	Аргумент > 0
	.IF GE	Аргумент ⩾ 0
	.IF LT	Аргумент < 0
	.IF LE	Аргумент ⩽ 0
Внешние условия	.IF B	Аргумент макрокоманды пропущен?
	.IF NB	Аргумент макрокоманды не пропущен?
Атрибут	.IF DF	Идентификатор аргумента определен?
	.IF NDF	Идентификатор аргумента не определен?
Символьная строка	.IF IDN	Два аргумента макрокоманды идентичны?
	.IF DIF	Два аргумента макрокоманды различны?

Заметим, что аргументы макрокоманд, заключенные в угловые скобки, например <A, B, C>, интерпретируются как один терм.

Примеры использования условного ассемблирования.

1. Использование условного ассемблирования для создания рекурсивного макроопределения:

```
A=0
.MACRO SUM X,Y
ADD     X,Y
A=A+1
.IF     NE,A-3
SUM     X,Y
.ENDC
.ENDM
```

2. Задача состоит в том, что макрогенератор должен вставить нужную команду в зависимости от расстояния до метки. Здесь целесообразно выделить три группы его значений: меньше или

равно 128 байтам, от 129 до 256 байт и, наконец, свыше 256 байт. В соответствии с этим должна быть вставлена команда SOB Rn, LOOP, DEC Rn/BNE LOOP или DEC Rn/BEQ .+6/JMP LOOP. Рассмотрите вопрос о том, когда эта макрокоманда будет генерировать неверное макрорасширение.

```
        .MACRO  SOBC    R0,LOOP
        SIZE=.-LOOP
;SOB?
        .IF     GT,126.-SIZE
        SOB     R0,LOOP
        .ENDC
;DEC & BRANCH?
        .IF     LT,124.-SIZE
        .IF     GE,252.-SIZE
        DEC     R0
        BNE     LOOP
        .ENDC
        .ENDC
;DEC & JUMP?
        .IF     LT,252.-SIZE
        DEC     R0
        BEQ     .+6
        JMP     LOOP
        .ENDC
;
        .ENDM
```

4.5.5. Блоки повторений, конкатенация и числовые аргументы

Иногда бывает необходимо повторить в исходном тексте программы некоторую последовательность кодов (блок кодов) несколько раз. Для этого используются фрагменты вида

```
.REPT   EXPR
code to be repeated
.ENDR
```

где EXPR — выражение, определяющее число повторений блока при ассемблировании. Допустим, что мы хотим сформировать в программе фиксированного размера блок пересылки, обладающий высоким быстродействием. Можно было бы решить эту задачу, написав последовательность команд, заканчивающуюся командой SOB, как это делалось в предыдущих примерах. Однако здесь мы стремимся достичь более высокого быстродействия за счет использования большего объема памяти, поэтому сделаем так:

```
.MACRO BLKMVF X,Y
MOV     #X,R0           ;X IS FIRST ADDRESS OF ARRAY 1
MOV     #Y,R1           ;Y IS FIRST ADDRESS OF ARRAY 2
.REPT   100.
MOV     (R0)+,(R1)+
.ENDR
.ENDM
```

Директива повторения может еще более эффективно применяться в сочетании с двумя другими средствами макроассемблера. Первое из них — конкатенация — заключается в объединении

фактических значений нескольких формальных параметров. В этом случае формальные параметры связаны апострофом ('), что означает «подменить формальные аргументы их значениями и слить эти значения воедино».

Второй прием заключается в использовании числового значения формального параметра. Аргументу, который интерпретируется как число, слева приписывается обратная косая черта (\). Сочетание приведенных механизмов позволяет получить такой интересный пример:

```
        .MACRO  INC A,B
        CNT     A,\B
        B=B+1
        .ENDM

        .MACRO  CNT A,B
A'B:    .WORD B
        .ENDM
```

Эти две макрокоманды вызываются так:

```
C=0
.REPT   5
INC     X,C
.ENDR
```

В результате получится следующее макрорасширение:

```
X0:     .WORD  0
X1:     .WORD  1
X2:     .WORD  2
X3:     .WORD  3
X4:     .WORD  4
```

Применение именно двух макрокоманд определяется тем, что формальный параметр B не может быть модифицирован внутри макрокоманды CNT. В ней число рассматривается только как набор символов, используемый для подстановки. Из приведенного примера также видно, что если имя макрокоманды совпадет с обозначением стандартного кода операции ассемблера, то оно пользуется приоритетом (сравните стандартную команду INC с написанной выше макрокомандой).

4.5.6. Эффективность макроассемблера. Реализация адресации системы VAX-11 на PDP-11

Макроассемблер, обладающий такими возможностями, как свойства вложенности макроопределений, условное ассемблирование и рекурсивный вызов, в качестве средства программирования обладает большей эффективностью, чем аппарат вызова подпрограмм. Его использование позволяет создавать программы, абсолютно точно соответствующие своему назначению. От подпрограмм, где анализируются те условия, которые *могут возникнуть* в процессе исполнения, сгенерированная программа отличается тем, что она точно соответствует условиям, *о существовании которых заранее не известно.*

Интересный пример генерации программы в процессе трансляции представляет моделирование подлинного индексного метода адресации системы VAX-11 на ЭВМ PDP-11. В VAX-11 используются те же методы адресации, что и PDP-11, за исключением метода 5. В любом из них, кроме регистрового, адреса операндов могут получать приращение при суммировании с константой, равной содержимому второго регистра, умноженному на размер операнда. Размер операнда соответственно равен 1, 2, 4 или 8 для байт, слов, двойных слов или значений с плавающей точкой одинарной точности и для учетверенных слов или значений с плавающей точкой двойной точности. Размер операнда определяется типом команды, аналогично тому, как в PDP-11 есть команды MOV и MOVB. Для указания индексирования к обычному описанию адреса добавляется в квадратных скобках индексный регистр. Например, при выполнении команды

```
INCW @(R2)+[R5]
```

адрес выбирается из ячейки, указанной в R2, к нему прибавляется 2, поскольку в соответствии с суфиксом W команда работает со словами, а результат складывается с удвоенным содержимым R5. Все это в итоге дает исполнительный адрес. Содержимое R5 при выполнении таких операций *не* модифицируется. В общем случае мы не будем считать индексными PC или другие регистры, вовлеченные в автоинкрементные или автодекрементные операции. При рассмотрении материала следующей главы станет ясна ценность использования подлинного индексного метода адресации.

Ниже приведена программа-транслятор, позволяющая использовать методы индексирования системы VAX-11 для ряда однооперандных команд PDP-11. Приводимый пример демонстрирует эффективность методов макроассемблирования, но, конечно, он не дает столь же эффективных программ, как на системе VAX-11. Поскольку квадратные скобки «[» и «]» ассемблером MACRO-11 не распознаются как разделители, вместо них будут употреблены угловые скобки «⟨» и «⟩». Ключом к использованию макрокоманд VAX является директива .HTYPE SYM, arg, которая определяет 6-битовое описание метода адресации и помещает его в переменную SYM. Тогда с помощью оператора & — логического «И» — в MACRO-11 можно выделить номера используемого регистра и метода адресации. Директива «.IRPC c, arg» повторяет заданный в ней блок кодов, заменяя в каждом повторении формальный параметр с символом из arg.

Предполагается, что читатель самостоятельно проанализирует приведенную ниже макрокоманду, проверив таким образом свое понимание возможностей MACRO-11, а также для того, чтобы убедиться в возможности реализации методов адресации, аналогичных применяемым в системе VAX-11. Заметим, что макрорасширение данной макрокоманды дает результирующие коды, которые сами себя модифицируют. Такое решение допустимо, когда

первостепенное значение имеет экономия памяти и программа эксплуатируется только одним программистом.

```
7         ;MACRO GENERATES UP TO 3 INSTRUCTIONS: A1, A2, AND A3
8         ;
9                 .MACRO    VAX     A,SYM1,A1,A2,A3
10                 .IRPC    XX,A
11                  .IF     EQ,RN&70-XX'0
12                   .IRPC  X,01234567
13                    .IF   EQ,SYM1&07-X
14                  A1
15                  A2
16                  A3
17                       .ENDC
18                      .ENDM
19                     .ENDC
20                    .ENDM
21                   .ENDM
22
23        ;Decode and implement "OPW DST <Rm>" instruction in PDP-11 code,
24        ; where OP is CLR, COM, INC, DEC, NEG or TST.  DST references Rn.
25        ;
26           .IRP OP, <CLR,COM,INC,DEC,NEG,TST>
27                .MACRO  OP'W  DST, INDEX, ?L
28        ;
29           .NTYPE RN, DST     ;RN ← 6-bit value = MODE, REGISTER of DST
30           .NTYPE RM, INDEX   ;RM ← 6-bit value = MODE, REGISTER of INDEX
31        ;.....................................................................
32        ;a) MODE 4? Generate "Pre-decrement Rn" instruction.
33        ;
34             VAX 4,RN,<     TST -(R'X)>
35        ;.....................................................................
36        ;b) MODE 1,2,3,4, or 5? Generate "Compute 2*Rm in offset" instrs.
37        ;
38             VAX 12345,RM,<     MOV R'X,L+2>,<     ASL L+2>
39        ;
40        ;c) MODE 6? Generate "Compute offset, add offset to base" instrs.
41        ;
42             VAX 6,RM,<     MOV R'X,T1>,<     ASL T1>,<     ADD T1,L+2>
43        ;.....................................................................
44        ;d) MODE 1,2, or 4? Generate OP 0(Rn). "0" will receive 2*Rm.
45        ;
46             VAX 124,RN,<L: OP 0(R'X)>
47        ;
48        ;e) MODE 3? Generate OP @#0. "0" contains PDP-11 address + 2*Rm.
49        ;
50             VAX 3,RN,<     ADD (R'X)+,L+2>,<L: OP @#0>
51        ;
52        ;f) MODE 5? Generate OP @#0. "0" contains PDP-11 address + 2*Rm.
53        ;
54             VAX 5,RN,<     ADD -(R'X),L+2>,<L: OP @#0>
55        ;
56        ;g) MODE 6? Generate OP DST. Offset will have 2*Rm added in,
57        ;             and old offset will later be restored by subtraction.
58        ;
59             VAX 6,RN,<L: OP DST>,<     SUB T1,L+2>
60        ;.....................................................................
61        ;h) MODE 2? Generate "Post increment Rn" instruction.
62        ;
63             VAX 2,RN,<     TST (R'X)+>
64        ;.....................................................................
65                .ENDM                       ;END FOR "OP'W" MACRO DEFINITION
66           .ENDM                            ;END FOR INITIAL .IRP OP, <...>
67                                            ;
68                                          ;*******************************
69 000000                                    CLRW    (R1) <R4>
   000000  010467  000006                        MOV R4,64$+2
   000004  006367  000002                        ASL 64$+2
   000010  005061  000000                   64$: CLR 0(R1)
70                                          ;*******************************
71 000014                                    COMW    (R2)+ <R5>
   000014  010567  000006                        MOV R5,65$+2
   000020  006367  000002                        ASL 65$+2
   000024  005162  000000                   65$: COM 0(R2)
   000030  005722                               TST (R2)+
```

```
72                                          ;*******************************
73 000032                                    TSTW   @(R4)+ <R5>
   000032  010567  000012                         MOV R5,66$+2
   000036  006367  000006                         ASL 66$+2
   000042  062467  000002                         ADD (R4)+,66$+2
   000046  005737  000000                  66$: TST @#0
74                                          ;*******************************
75 000052                                    INCW   -(R3) <R4>
   000052  005743                                 TST -(R3)
   000054  010467  000006                         MOV R4,67$+2
   000060  006367  000002                         ASL 67$+2
   000064  005263  000000                  67$: INC 0(R3)
76                                          ;*******************************
77 000070                                    NEGW   @-(R0) <R2>
   000070  010267  000012                         MOV R2,68$+2
   000074  006367  000006                         ASL 68$+2
   000100  064067  000002                         ADD -(R0),68$+2
   000104  005437  000000                  68$: NEG @#0
78                                          ;*******************************
79 000110                                    CLRW   A <R4>
   000110  010467  000056                         MOV R4,T1
   000114  006367  000052                         ASL T1
   000120  066767  000046  000002.                ADD T1,69$+2
   000126  005067  000036                  69$: CLR A
   000132  166767  000034  177770                 SUB T1,69$+2
80                                          ;*******************************
81 000140                                    DECW   A(R4) <R0>
   000140  010067  000026                         MOV R0,T1
   000144  006367  000022                         ASL T1
   000150  066767  000016  000002                 ADD T1,70$+2
   000156  005364  000170'                 70$: DEC A(R4)
   000162  166767  000004  177770                 SUB T1,70$+2
82                                          ;*******************************
83 000170  000052                           A:     52
84 000172  000000                           T1:    0
85         000001'                                 .END
```

Условное ассемблирование довольно широко употребляется при генерации операционных систем. При этом в качестве генератора можно использовать программу на языке Ассемблера, задавая как параметры характеристики тех условий, в которых операционной системе предстоит работать. В результате после ассемблирования будет получена версия операционной системы, соответствующая конкретным задачам и конфигурации вычислительной системы. Генераторы такого типа не выполняют анализа объема оперативной памяти и дополнительных возможностей ЭВМ; эта информация вводится как параметры.

Еще один пример эффективности применения макросредств — полностью параметризованные макропрограммы. Командами в них могут быть макрокоманды или макроопределения, выполненные на основе макрокоманд, определенных ранее. В системах такого рода пользователю не известно соответствие между макрокомандами и машинными командами. В самом деле, если новые макроопределения основываются только на уже известных макрокомандах, то программисту не нужно ничего знать о машине, на которой ему приходится работать.

Классическим примером осуществления такой схемы служит реализация языка SNOBOL4. Он представляет собой результат макрогенерации и требует лишь, чтобы каждая из его макрокоманд была реализована на той ЭВМ, для которой создается версия интерпретатора. Если это условие выполнено, то система SNOBOL4 может быть ассемблирована в работающий языковый процессор.

УПРАЖНЕНИЯ

1. Напишите подпрограмму умножения двух целых со знаком, результат — целое двойной длины. Вызов подпрограммы может быть таким:

```
MOV     #MIER,R0        ;MULTIPLIER ADDRESS IN R0
MOV     #MAND,R1        ;MAND ADDRESS IN R1
JSR     PC,MULT         ;CALL MULTIPLY
```

2. Напишите программу умножения в PIC-форме так, чтобы можно было использовать вызов

```
JSR     R5,MULT
.WORD   M1-ADDRESS,M2-ADDRESS
```

3. Проверьте, выполняет ли на самом деле программа в параграфе 4.3.4б требуемое перемещение всех указателей в таблице.

4. Напишите PIC-программу, которая бы обнуляла все ячейки памяти, расположенные до ее кодов (т. е. если программа расположена с ячейки 10 000, то обнулить ячейки 0—7777).

5. Напишите программу, вызывающую подпрограмму из упр. 4.

6. Имеются основная программа и подпрограмма:

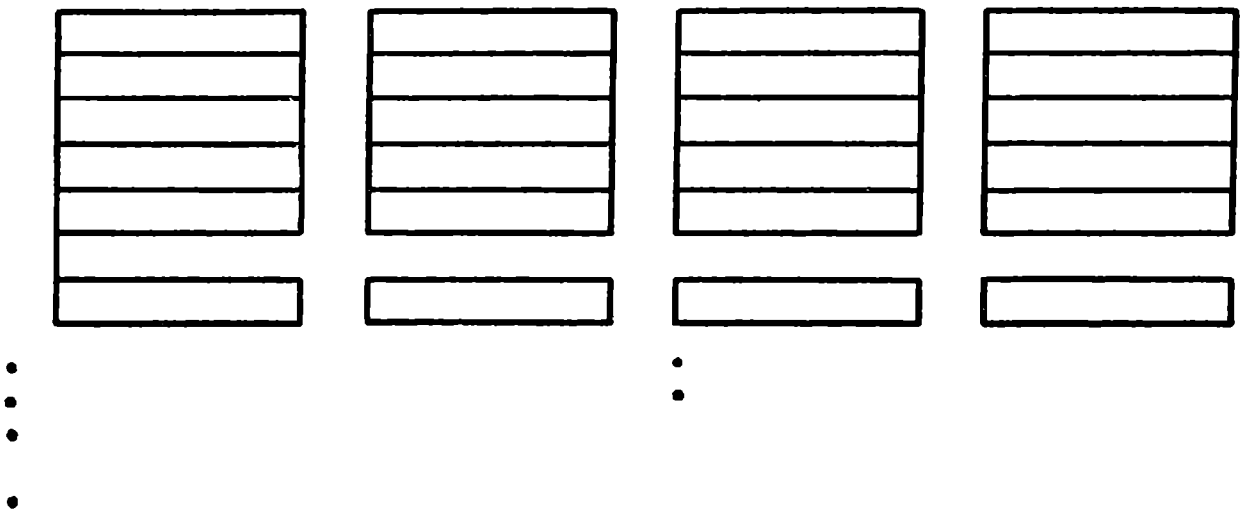
```
        .=1000
BEGIN:  MOV     #BEGIN,SP
        MOV     #1,R0
        JSR     R0,SUB
        .WORD   3
```

```
        .=2000
SUB:    MOV     (R0)+,R1
        DEC     R1
        TST     R1
        BEQ     RETURN
        JSR     R0,SUB
        .WORD   1
RETURN: RTS     R0
```

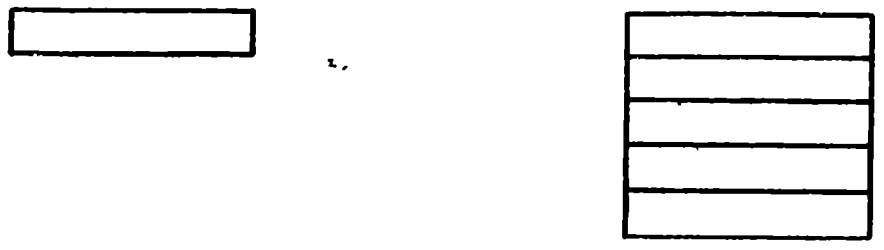
а. Запишите в бланк содержимое указателя стека и самого стека процессора до выполнения программы. (Примечание. В этой части упражнения и далее для обозначения неизвестного значения употребляется символ «?».)

б. Запишите в бланк содержимое указателя стека и самого стека процессора сразу после выполнения команды MOV #1, R0 в основной программе.

в. Запишите в бланк содержимое указателя стека и самого стека процессора *после выполнения каждой стековой операции*, т. е. после записи или чтения. Обозначьте также каждую используемую ячейку стека. Имеется несколько бланков, но вы можете не использовать их все; начинайте заполнение с левого.



г. Запишите в бланк содержимое указателя стека и самого стека процессора по окончании выполнения основной программы.

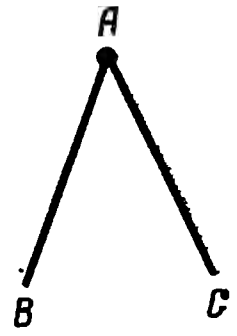


7. Ниже приведена рекурсивная подпрограмма для обработки данных, имеющих структуру в виде *двоичного дерева.* Двоичное дерево состоит из *узлов*

```
        .MCALL .EXIT .TTYOUT      ;RT-11 SYSTEM MACROS
TRAVRS:
        MOV     #TRAVRS,SP        ;INITIALIZE THE STACK
        MOV     ROOT,R1           ;ROOT CONTAINS ADDRESS OF ROOT NODE
        JSR     R5,GO
        .EXIT                     ;BACK TO RT-11 MONITOR
GO:
        TST     R1                ;ZERO MEANS NO SUCCESSOR
        BEQ     RETURN
        ;
        .TTYOUT 2(R1)             ;OUTPUT  2(R1) ON TERMINAL (SEC. 7.7)
        ;
        MOV     R1,-(SP)          ;SAVE POINTER
        MOV     (R1),R1           ;ADDRESS OF LEFT SUCCESSOR
        JSR     R5,GO
        ;
        MOV     (SP)+,R1          ;RECOVER POINTER
        MOV     4(R1),R1          ;ADDRESS OF RIGHT SUCCESSOR
        JSR     R5,GO
RETURN:
        RTS     R5
        ;
        .MACRO NODE LEFT,NAME,RIGHT
NAME:   .WORD   LEFT              ;LEFT SUCCESSOR IDENTITY
        .ASCII  /NAME/            ;CONTENTS OF THIS NODE
        .BYTE   0
        .WORD   RIGHT             ;RIGHT SUCCESSOR IDENTITY
        .ENDM
        ;
ROOT:   .WORD   A                 ;ROOT OF THE TREE
        NODE    B,A,C
        NODE    D,B,0
        NODE    E,C,F
        NODE    0,D,0
        NODE    0,E,G
        NODE    H,F,J
        NODE    0,G,0
        NODE    0,H,0
        NODE    0,J,0
        ;
        .END    TRAVRS
```

и *ветвей*, каждая ветвь соединяет два узла. Хотя такого рода структуры не изучаются в данной книге, эта программа представляет интерес по нескольким соображениям. Наиболее важное из них то, что она демонстрирует значение и мощность методов рекурсивного программирования. Любое двоичное дерево состоит из корня, а также правого и левого поддерева, выходящих из этого корня. *Каждое поддерево само является деревом*, из чего следует рекурсия.

а. Нарисуйте двоичное дерево. Наверху поставьте точку, представляющую корень «А», затем, спускаясь вниз, рисуйте правые и левые ветви, соединяющие каждый узел с его соседом. Например, первые две ветви выглядят так:



б. При выполнении подпрограмма должна «проходить» по дереву, т. е. по одному разу «зайти» в каждый узел и, например, распечатать его содержимое. Для прохода имеются три алгоритма:

I. *Последовательный.* Посещение узлов выполняется в следующем порядке: для любого узла V дерева все узлы левого поддерева посещаются *до* узла V, а все узлы правого поддерева — *после* этого. В более компактной форме:

в данном корне

1) последовательно посетить узлы левого поддерева;
2) посетить корень;
3) последовательно посетить узлы правого поддерева.

II. *После данного.*

В данном корне

1) последовательно пройти по левому поддереву;
2) последовательно пройти по правому поддереву;
3) посетить корень.

III. *До данного*

В данном корне

1) посетить корень;
2) последовательно пройти по левому поддереву;
3) последовательно пройти по правому поддереву.

в. Просмотрите программу. Какой алгоритм использован в ней? Можете ли вы изменить программу с целью применить один из двух других алгоритмов? Учтите, что работа системных макрокоманд TTYOUT и .EXIT описана в гл. 7.

8. Напишите расширение макровызова

```
SWITCH BUF1,BUF2,SPARE,10
```

в соответствии с макроопределением в параграфе 4.5.3.

9. Перепишите макроопределение для макрокоманды SWITCH так, чтобы для промежуточного хранения требовался массив TEMP длиной только в одно слово.

10. Напишите макрокоманду BSS X, которая бы резервировала блок ячеек длиной X байт.

11. Разработайте рекурсивную макрокоманду, которая бы выполняла умножение при помощи сдвига и сложения.

БИБЛИОГРАФИЯ

В связи с тем, что позиционно-независимое кодирование не сильно отличается от кодирования с использованием регистра базы, читателям, которые интересуются этим вопросом, для дальнейшего чтения могут быть рекомендованы Stone H. S. (1975), Gear C. W. (1974), Abrams M. D. и Stein P. G. (1973), Mauer W. D. (1968). В этих книгах рассматриваются вопросы, связанные с подпрограммами и с вызовом подпрограмм. Для изучения такой тематики, как вызов по значению, вызов по имени и вызов по адресу, следует читать Wegner P. (1968) и Ralston A. (1971).

Классическое изложение методов рекурсии приведено в небольшой книге Barron D. W. (1968). Особенность подхода, выбранного автором, отражена в первых фразах книги: «Было замечено, что «если бы вычислительные машины существовали в средние века, то программисты сжигали бы друг друга по обвинению в ереси»... (и что) одной из основных ересей была бы вера (или неверие) в рекурсию». Общая проблема ассемблирования для машин, имеющих команды с ограниченным диапазоном доступа в память, поднятая в параграфе 4.5.4 при описании макрокоманды SOBC, в деталях рассматривается в Szymanski T. G. (1978).

Глава 5

СТРУКТУРЫ ДАННЫХ И РАБОТА С НИМИ

В предыдущих главах при изложении таких понятий, как команды, форматы информации и элементов программирования, составляющих основу методов машинной обработки информации, употреблялись *простейшие* элементы данных (биты, байты, слова) и для работы с ними использовались *простые* арифметические и логические функции (сложение, логическое ИЛИ и т. п.). В этой главе читателю предлагается детализация более сложных арифметических и логических операций над данными, также имеющими более сложную структуру.

5.1. ОБРАБОТКА БАЙТ И КОДЫ СИМВОЛОВ

Процессор PDP-11 включает полный набор команд для обработки байтовых операндов. Поскольку в PDP-11 допускается байтовая адресация, т. е. имеется возможность выполнения операций непосредственно над байтами, эту ЭВМ можно интерпретировать и как процессор, оперирующий байтами, и как процессор, ориентированный на слова. Все одно- и двухоперандные команды, за исключением ADD и SUB, могут быть выполнены в байтовом режиме; для этого к мнемоническому коду операции просто прибавляется символ «B», например MOVB, DECB.

Ценность вычислительной машины, имеющей байтовую адресацию, состоит в том, что она позволяет обращаться к единице информации, представляющей отдельный символ. Используемый в PDP-11 байт содержит 8 бит, что создает возможность для представления набора из 256 различных символов, подмножеством которого является семибитовый код ASCII. Это подмножество образует набор алфавитно-цифровых символов PDP-11, и наличие его обеспечивает выполнение обработки символьной информации.

При использовании байтовых операций для работы с символами или байтами следует иметь в виду 2 обстоятельства. Во-первых, в регистровом методе адресации употребляется младший байт указанного в команде регистра Во-вторых, при применении в байтовых командах прямого автоинкрементного или автодекрементного метода адресации значение указанного в команде

регистра изменяется на 1. Следовательно, для увеличения значения регистра на 4 может быть использована команда

```
CMP     (R0)+,(R0)+
```

Та же команда, выполненная в байтовом режиме,

```
CMPB    (R0)+,(R0)+
```

увеличит значение регистра только на 2.

Для удобства работы с символьными данными в MACRO-11 включена директива .BYTE, которая позволяет запоминать 8-битовые значения в последовательно расположенных байтах. Кроме того, имеется директива .ASCII, полезная при ассемблировании строки символов в коде ASCII в их байтовое представление, а также директива .EVEN, используемая для настройки счетчика адреса на границу слова.

Если символу в поле операнда предшествует знак апострофа, это значит, что при трансляции в соответствующий байт будет помещено двоичное представление этого символа в коде ASCII. Аналогично в слово будет записано представление двух символов, если им предшествует кавычка, причем каждый из них займет один 8-битовый байт. Например, запись "AB приведет к размещению кодов букв A и B в одном слове, где код символа A будет храниться в младшем (четном), а код B — в старшем (нечетном) байтах. С помощью этих средств засылка кода символа A в R0 может быть выполнена следующей командой:

```
MOVB    #'A,R0
```

а для пересылки в R0 строки AB команда имеет вид

```
MOV     #"AB,R0
```

(Обратите внимание на употребление символа «#». Без него последовательность символов будет интерпретирована как адрес!)

5.2. АРИФМЕТИЧЕСКИЕ ОПЕРАЦИИ С ФИКСИРОВАННОЙ ТОЧКОЙ

Из арифметических операций мы пока употребляем только команды сложения и вычитания слов одинарной длины. Однако при выполнении операций с фиксированной точкой более употребительны форматы представления чисел, которые для увеличения точности занимают больше одного слова. Арифметические операции над такими числами требуют дополнительных команд, использующих биты *переноса* и *переполнения*. При этом выполнение операций умножения и деления обеспечивается с помощью *сдвигов*.

5.2.1. Перенос и переполнение

В гл. 3 при рассмотрении арифметических операций над двоичными и восьмеричными числами нами сознательно был опущен один вопрос, а именно работа с большими числами, т. е. числами, значения которых превышают арифметические возможности машины. Допустим, что существует ЭВМ с 4-битовым словом, отрицательные числа в ней представляются в дополнительном коде и она может выполнять сложение и вычитание. Как показано в табл. 5.1, диапазон представляемых в этой машине чисел заключается в пределах от +7 до —8.

Таблица 5.1. **Представление четырехбитовых чисел**

Двоичное значение	Десятичное значение
0111	+7
0110	+6
0101	+5
0100	+4
0011	+3
0010	+2
0001	+1
0000	0
1111	—1
1110	—2
1101	—3
1100	—4
1011	—5
1010	—6
1001	—7
1000	—8

В двоичной форме сложение десятичных чисел 3 и 4 будет выглядеть так:

```
3       0011
4       0100
--      ----
7       0111
```

Сложение же чисел 3 и 5 даст в результате —8:

```
3       0011
5       0101
--      ----
8       1000 = -8
```

Суть этого явления заключается в том, что при сложении произошел *перенос* в знаковый разряд и получился ошибочный результат. Следовательно требуется, чтобы арифметическое устройство вырабатывало признак *переполнения*, свидетельствующий о том, что результат арифметической операции попал в старший бит слова (MSB) или в знаковый разряд. Термины *перенос* и *переполнение* используются для близких понятий, поэтому требуются определенные уточнения.

По мере изучения системы команд читатель узнает действительное различие между ними. Сейчас же рассмотрим, как и

когда вырабатываются признаки переноса и переполнения при выполнении команды сложения ADD.

1. ПЕРЕНОС вырабатывается тогда, когда происходит перенос из позиции MSB слова.

2. ПЕРЕПОЛНЕНИЕ вырабатывается в том случае, когда оба слагаемых имеют один знак, а результат — противоположный.

Смысл различия между ПЕРЕНОСОМ и ПЕРЕПОЛНЕНИЕМ состоит в том, что признак ПЕРЕНОС используется для выполнения *операций с увеличенной точностью*, а признак ПЕРЕПОЛНЕНИЕ нужен только для определения правильности или неправильности выполнения арифметической операции.

В табл. 5.2 на примере 4-битовой машины демонстрируется различие между переносом и переполнением. Неверный результат вырабатывается только при значении бита переполнения, равном единице.

Таблица 5.2. Установка бит V и C при операциях над четырехбитовыми числами

Десятичное	Двоичное	Перенос из MSB	Неверный знак	Значение		Результат
				V	C	
1	0001					
+2	0010	нет	нет	0	0	верный
3	0011					
5	0101					
+6	0110	нет	да	1	0	ошибочный
11	1011					
−6	1010					
+7	0111	да	нет	0	1	верный
1	0001					
−6	1010					
−6	1010	да	да	1	1	ошибочный
−12	0100					

5.2.2. Сложение и вычитание с повышенной точностью

При анализе выполнения арифметических операций с двойной точностью становится яснее значение бита переноса. Число с двойной точностью, в данном случае 8-битовое число, хранится в двух 4-битовых словах. Следовательно, таким образом могут быть представлены числа, десятичные значения которых заключены в пределах от —128 до 127, поскольку для них используются семь разрядов плюс один знаковый.

Для выполнения сложения с двойной точностью служит следующий алгоритм:

1. Сложить младшие 4 бита (*младшее слово*).
2. Сложить старшие 4 бита (*старшее слово*) и бит переноса, если он равен единице.
3. По значению бита переноса, выработанному на шаге 2, определяют корректность результата сложения.

Таблица 5.3. **Примеры сложения с двойной точностью**

Десятичное	Двоичное	Сложение младших слов		Сложение старших слов		Результат
		V	C	V	C	
49	0011 0001					
+50	0011 0010	0	0	0	0	верный
99	0110 0011					
53	0011 0101					
+70	0100 0110	1	0	0	0	верный
123	0111 1011					
60	0011 1100					
+70	0100 0110	0	1	1	0	ошибочный
130	1000 0010					
26	0001 1010					
+26	0001 1010	1	1	0	0	верный
52	0011 0100					
—70	1011 1010					
—70	1011 1010	1	1	1	1	ошибочный
—140	0111 0100					
—70	1011 1010					
80	0101 0000	0	0	0	1	верный
10	0000 1010					

Пример выполнения сложения с двойной точностью из табл. 5.3 показывают, что значение бита переноса, полученное в результате операции над младшими словами, используется как дополнительное слагаемое в операции над старшими. При сложении младших слов значение признака переполнения игнорируется, поскольку на этом этапе переполнение вполне допустимо при выполнении арифметической операции с двойной точностью. Это значение учитывается на шаге сложения старших слов.

Диапазон чисел, которые могут быть представлены в одном слове, из-за его малой длины сильно ограничен, поэтому осуществление операций с двойной точностью является характерной особенностью мини-ЭВМ. В связи с этим PDP-11 имеют специальные команды сложения и вычитания бит переноса.

Команда	Символическая форма	Значение
Прибавить перенос	ADC X	(X) ← (X) + (C)
Вычесть перенос	SBC X	(X) ← (X) — (C)

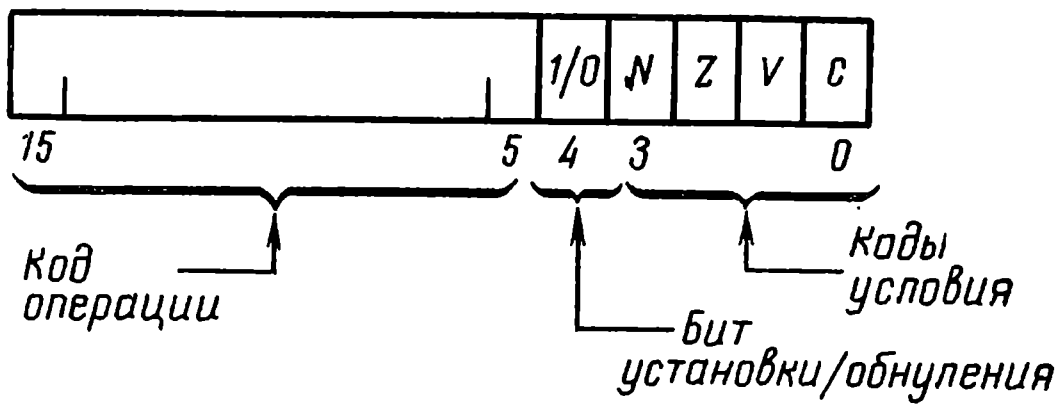
Обозначение (C) соответствует значению бита C — одного из кодов условия. В системе PDP-11 присвоение значений битам C и V выполняется в результате таких операций, как ADD, SUB, COM или NEG. Таким образом, завершение арифметической операции характеризуется не только значениями бит Z и N, но и кодов условий C и V.

```
        ADD     A,B
        .
A:      .WORD   121354
B:      .WORD   134201
```

Операция сложения из следующего примера приведет к получению ошибочного результата — 055555, слово состояния процессора при этом получит значение 000003 (см. рис. 3.1). Выполняемая затем команда, это может быть, например, MOV, обнулит бит V, установит в единицу биты N и Z, но оставит неизменным содержимое бита C *. В некоторых случаях может возникнуть необходимость в непосредственном изменении значений кодов условий, их обнулении или установке. В системе PDP-11 для этой цели существует такой набор команд:

Команда	Символическая форма	Значение
Обнулить бит C	CLC	(C) ← 0
Обнулить бит V	CLV	(V) ← 0
Обнулить бит Z	CLZ	(Z) ← 0
Обнулить бит N	CLN	(N) ← 0
Установить бит C в 1	SEC	(C) ← 1
Установить бит V в 1	SEV	(V) ← 1
Установить бит Z в 1	SEZ	(Z) ← 1
Установить бит N в 1	SEN	(N) ← 1

Они имеют следующий формат:



С помощью команд такого набора можно обнулить или установить в единичное состояние произвольную комбинацию бит условий. Для этого несколько кодов операций следует объединить логической операцией «ИЛИ», которая в Ассемблере обозначается знаком «!», например CLC ! CLV. Используя возможности, которые дают для установки в 1 и обнуления бит C и V команды

* Команда MOV устанавливает или очищает коды условий Z и N в зависимости от содержимого операнда-источника. — *Примеч. ред.*

ADD и NEG, операции двойной точности могут быть выполнены следующим образом:
сложение

```
ADD     A0,B0           ;ADD LOW-ORDER WORDS
ADC     B1              ;ADD CARRY INTO HIGH-ORDER WORD
ADD     A1,B1           ;ADD HIGH-ORDER WORDS
```

дополнение

```
NEG     A1              ;2'S COMPLEMENT HIGH-ORDER WORD
NEG     A0              ;2'S COMPLEMENT LOW-ORDER WORD
SBC     A1              ;PROPAGATE CARRY
```

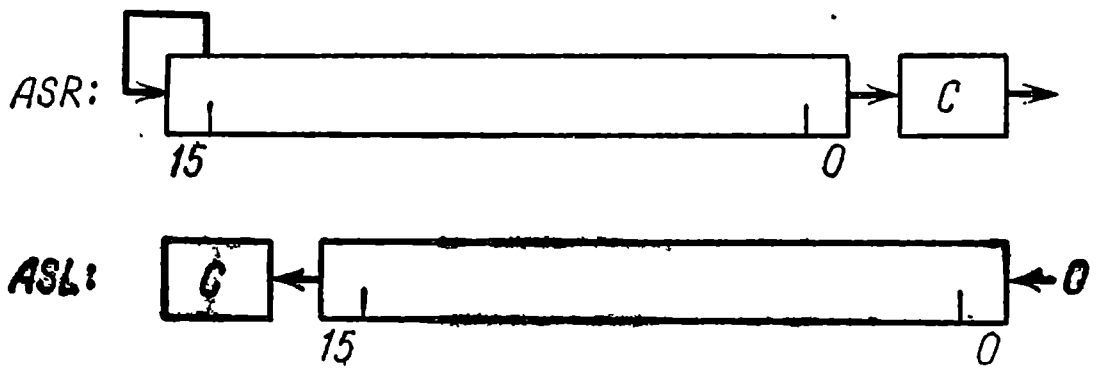
(Команда NEG устанавливает бит переноса C в 1, если результат дополнения до двух не равен нулю, и в 0 — во всех остальных случаях.) Операции с двойной и большей точностью используются для умножения чисел с одинарной точностью (результат получается с двойной точностью), а также для выполнения операций с плавающей точкой, см. разд. 5.2.4.

5.2.3. Сдвиг

Помимо команд, позволяющих работать с данными на уровне байт или слов, необходимы и такие, которые дают возможность оперировать отдельными битами внутри слова. Поэтому PDP-11 включает команды сдвига, циклического сдвига и логических операций (см. разд. 5.3).

Команды сдвига и циклического сдвига принадлежат к группе одноопернадных команд. Первые из них выполняют арифметический сдвиг на один разряд влево или вправо, причем бит C применяется в качестве расширения регистра, в котором выполняется сдвиг. Так, при сдвиге влево MSB сдвигается в бит C, а все остальные смещаются на один бит влево, позиция младшего бита слова (LSB) заполняется нулем. Сдвиг вправо отличается от описанного тем, что в MSB заносится повторение значения знака, а LSB сдвигается в бит C. Сдвиг вправо не приводит к изменению знака. Любую операцию сдвига можно рассматривать как масштабирование данных с коэффициентом 2.

Графически команды арифметического сдвига можно проиллюстрировать следующим образом:

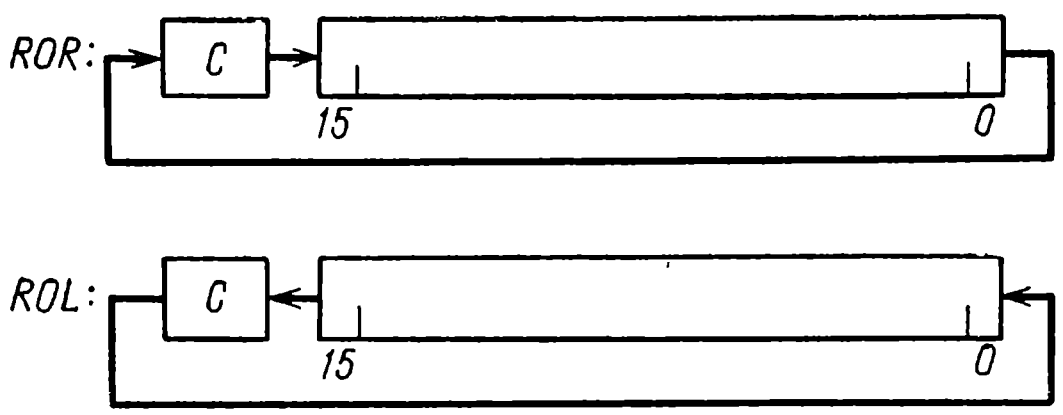


В соответствии с полученным результатом формируются значения бит N, Z и C. Биту V присваивается значение сложения по модулю 2 значений бит N и C, что дает возможность установить

возникновение арифметического переполнения, когда знак результата отличается от знака операнда.

Команды циклического сдвига работают со своими операндами и с битом C так, как если бы они вместе образовывали 17-разрядный «циклический буфер». При циклическом сдвиге влево MSB сдвигается в бит C, бит C сдвигается в LSB, а все остальные сдвигаются на одну позицию влево. При циклическом сдвиге вправо выполняются обратные действия. Вместе эти операции позволяют анализировать — модифицировать отдельные биты.

Наглядно команды циклического сдвига могут быть представлены так:



Циклические сдвиги порождают коды условий в соответствии с теми же правилами, что и для арифметических сдвигов.

Все четыре инструкции сдвигов записываются следующим образом:

Команда	Символическая форма	Функция
Арифметический сдвиг вправо	ASR X	(C) ← (X [0]) (X [14 : 0]) ← (X [15 : 1]) (X [15]) ← (X [15])
Арифметический сдвиг влево	ASL X	(C) ← (X [15]) (X [15 : 1]) ← (X [14 : 0]) (X [0]) ← 0
Циклический сдвиг вправо	ROR X	(X [14 : 0]) ← (X [15 : 1]) (X [15]) ← (C) (C) ← (old X [0])
Циклический сдвиг влево	ROL X	(X [15 : 1]) ← (X [14 : 0]) (X [0]) ← (C) (C) ← (old X ([15])

При использовании ЭВМ с расширенным набором команд (EIS) несколько арифметических или циклических сдвигов реализуются одной командой ASH и ASHC. Так, инструкция ASH SRC, Rn в соответствии со значением SRC осуществляет сдвиг максимум на 31 разряд влево или на 32 разряда вправо. Если SRC *положительно*, то выполняется сдвиг *влево*, если *отрицательно* — вправо. При сдвиге влево в младший бит регистра Rn заносится 0, в то время как при сдвиге вправо освобождающиеся биты заполняются значением, совпадающим со знаком числа. Команда

ASHC SRC, Rn сдвигает 32-разрядное слово в Rn и Rn + 1, где n четное; SRC используется так же, как в ASH. Если n нечетное, то может быть выполнен *только* циклический сдвиг не более чем на 16 разрядов в регистре Rn. Таким образом, команда ASHC подобна ASR и ASH, но работает с двойной длиной, и количество сдвигов может быть задано в операнде.

5.2.4. Умножение

Сочетания операций сдвигов и сложения позволяют осуществлять умножение чисел без знака, например:

```
118(10)=   001 110 110 (2)
 11(10)=   000 001 011 (2)
          -------------
              1 110 110
             11 101 10
            000 000 0
           1 110 110
          -------------
          10 100 010 010 (2) = 1298 (10)
```

Этот пример может быть запрограммирован следующим образом:

```
        CLR   R0          ;CLEAR ACCUMULATOR
        MOV   MIER,R1     ;GET MULTIPLIER
        MOV   MAND,R2     ;GET MULTIPLICAND
        MOV   #16.,R3     ;SET LOOP COUNT
        ROR   R1          ;SHIFT LOW BIT OF MULTIPLIER INTO C
LOOP:   BCC   NADD        ;NO ADD IF LOW BIT OF MULTIPLIER ZERO
        ADD   R2,R0       ;ADD MULTIPLICAND TO PARIAL PRODUCT
NADD:   CLC               ;CLEAR CARRY AS PARTIAL PRODUCT UNSIGNED
        ROR   R0          ;SHIFT HIGH WORD OF PARTIAL PRODUCT RIGHT
        ROR   R1          ;SHIFT LOW  WORD OF PARTIAL PRODUCT RIGHT
        SOB   R3,LOOP     ;LOOP. SOB DOES NOT ALTER C BIT
```

В регистрах R0 и R1 находится 32-разрядный результат. Однако в общем случае отрицательные числа, представленные в форме дополнения до 2, так умножать нельзя. Если вместо данного алгоритма использовать алгоритм Бута (Booth), то можно умножать и положительные, и отрицательные числа. Точно обосновать этот алгоритм довольно сложно, но можно продемонстрировать его основные принципы на простом примере.

Положим множимое равным —41. В форме двоичного 16-разрядного числа с дополнением до 2 оно будет выглядеть так:

```
1 111 111 111 010 111(2) = 177727(8) = -41(10)
```

Теперь, переписав его в виде сумм и разностей целых, содержащих только единицы (на основе выражения $(2 ** n - 1)$, получим следующее двоичное представление числа —41:

```
  1 111 111 111 111 111   -    111 111
+             11 111      -      1 111
+                                  111
```

Заменив (2 ** n —1) на (2 ** n) — (1), можем записать

```
(10 000 000 000 000 000-1)- (1 000 000-1)
+                 (100 000-1)-    (10 000-1)
.+                                 (1 000-1)
```

Затем, оставляя только ненулевые разряды, получим

```
 10 000 000 000 000 000    -      1 000 000
+                 100 000    -         10 000
+                   1 000    -              1

-    2**16  -2**6 +2**5 -2**4 +2**3 -2**0
```

Пренебрегая значением 2 ** 16 и заметив, что 2 ** 5 = 2 * 2 ** 4, будем иметь

```
1 111 111 111 010 111    = -2**6 +2**4 +2**3 -2**0
```

Теперь произведение (1 111 111 111 010 111) * m, где m — множитель, можно представить как $-(2 ** 6) * m + (2 ** 4) m +$ $+ (2 ** 3) * m - m$. Если $m = 75$, то $-41 * 75 = -3075 - 64 * 75 +$ $+ 16 * 75 + 8 * 75 - 75$.

При работе этого алгоритма выполнение операций сложения и вычитания требуется только тогда, когда в множимом не совпадают значения в соседних разрядах, т. е. имеется переход от 0 к 1 или от 1 к 0. Этот факт используется в *модифицированном алгоритме Бута*, в котором последовательности сложений и вычитаний генерируются автоматически. Допустим, что существует 16-разрядное в форме дополнения до 2 множимое mr, младший бит которого обозначен как $y(0)$, а старший — $y(15)$. Для $i = 0, 2, 4...$ будем анализировать значения $y(i-1)$, $y(i)$ и $y(i+1)$. Затем по табл. 5.4 определим действия, которые нужно выполнить над промежуточным результатом. R обозначает сдвиг промежуточного результата вправо с увеличением количества разрядов, занимаемых знаком, A и S — суммирование множителя с промежуточным результатом или вычитание его из промежуточного результата. Заметим, что $y(-1) = 0$.

Таблица 5.4. Декодирование умножения для модифицированного алгоритма Бута

y(i+1)	y(i)	y(i-1)	*Сдвиг промежуточного результата (ов)-R; прибавление множителя-A; вычитание-S*
0	0	0	R,R
0	0	1	A,R,R
0	1	0	A,R,R
0	1	1	R,A,R
1	0	0	R,S,R
1	0	1	S,R,R
1	1	0	S,R,R
1	1	1	R,R

Применим этот алгоритм к $mr = -41$:

```
1 111 111 111 010 111.0 (2) = -41(10)
                    -- -      -> S,R,R
                  - --        -> R,A,R
                ---           -> A,R,R
              -- -            -> S,R,R
           - --               -> R,R
         ---                  -> R,R
     -- -                     -> R,R
- --                          -> R,R
```

Тогда будет сгенерирована следующая цепочка: S, R, R, R, A, R, A, R, R, S, R, R, R, R, R, R, R, R, R, R. Учитывая, что выполнение r сдвигов вправо эквивалентно умножению на $2 ** (-r)$, произведение $-41 * m$ можно переписать в виде

```
[[[[-m]*2**(-3) +m]*2**(-1) +m]*2**(-2) -m]*2**(-10)
= -m*2**(-16) +m*2**(-13) +m*2**(-12) -m*2**(-10)
```

Однако для того, чтобы компенсировать сдвиг вправо, результат должен быть умножен на $2 ** 16$. В итоге получим то же выражение, что и прежде: $-41 * m = -(2 ** 6) * m + (2 ** 4) * m + (2 ** 3) * m - (2 ** 0) * m$.

Ниже описывается эффективная реализация модифицированного алгоритма Бута с помощью таблицы переходов. Прежде всего умножим mr на 2, чтобы младший бит, обозначенный как $y(-1)$, стал равен 0. Множимое содержится в 17-разрядном регистре (код условия C и R1), а начальное значение промежуточного результата — в R0. Объединение регистра $R0$ с битом C и регистром $R1$ можно интерпретировать как 33-разрядный «регистр». На каждом шаге алгоритма его содержимое может *дважды* сдвигаться командами ROR или ASR и множитель m может суммироваться или вычитаться из 16 старших разрядов такого регистра.

Поскольку две команды — ASL R4, требуемая для доступа в таблицу TABLE, и команда ADD или SUB — по условию могут изменить значение бита переноса, то *первая* команда ROR R1 должна предшествовать соответствующим командам ASL, ADD или SUB так, чтобы бит промежуточного результата, находящийся в C, сохранялся в 15-м разряде регистра R1. Поскольку *вторая* команда ASR R0 всегда является последней в каждом модуле, вызываемом для использования таблицы переходов, она должна быть помещена в основную программу с меткой JMPRET.

Заметим, что JMP может быть замещена инструкцией JSR, но первая более эффективна, в частности при возврате. Заключительная команда ROR R1, помеченная меткой FIN, необходима для компенсации ROL R1, выполненной для первоначального умножения $2 * mr$.

```
MULT:
        MOV   MIER,R1     ;R1 ← MULTIPLIER
        MOV   MAND,R2     ;R2 ← MULTIPLICAND
        MOV   #8.,R3      ;LOOP 8 TIMES
        ;
                          ;R0,C,R1 FORM 33-BIT SHIFT REGISTER
        CLC               ;CLEAR CARRY
        ROL   R1          ;2*MULTIPLIER IN C,R1.
        CLR   R0          ;ZERO PARTIAL PRODUCT HIGH WORD
        ;
;       *** INNER LOOP, EXECUTED 8 TIMES **********************************
LOOP:
        MOV   R1,R4       ;DUPLICATE R1
ROR1:   ROR   R1          ;SHIFT R1 RIGHT, SAVING CARRY IN BIT 15 OF R1
        ;
        BIC   #17770,R4   ;MASK OUT UPPER 13 BITS OF MR FOR BYTE OFFSET
        ASL   R4          ;FORM WORD OFFSET
        JMP   @TABLE(R4)  ;JUMP TO CORRECT MODULE
        ;
JMPRET: ASR   R0          ;FINAL "ASR" OF EACH SEQUENCE DONE HERE
        SOB   R3,LOOP     ;CARRY NOT AFFECTED BY 'SOB'
;       *** END OF INNER LOOP *********************************************
        ;
FIN:    ROR   R1          ;COMPENSATE FOR INITIAL 'ROL R1'
        MOV   R0,HIGANS   ;32-BIT PRODUCT
        MOV   R1,LOWANS   ; STORED
        RTS   PC          ;RETURN


TABLE:                    ;JUMP TABLE
        RR
        ARR
        ARR
        RAR
        RSR
        SRR
        SRR
        RR
        ;
RR:     ASR   R0          ;1'ST 'ROR' AT "ROR1" IN MAIN, 1'ST 'ASR' HERE
        ROR   R1          ;2'ND 'ROR' HERE,2'ND 'ASR' AT "JMPRET" IN MAIN
        BR    JMPRET
        ;
ARR:    ADD R2,R0         ;ADD MULTIPLICAND TO PARTIAL PRODUCT
        ASR   R0
        ROR   R1
        BR    JMPRET
        ;
RAR:    ASR   R0
        ROR   R1
        ADD   R2,R0       ;ADD MULTIPLICAND TO PARTIAL PRODUCT
        BR    JMPRET
        ;
RSR:    ASR   R0
        ROR   R1
        SUB   R2,R0       ;SUBTRACT MULTIPLICAND FROM PARTIAL PRODUCT
        BR    JMPRET
        ;
SRR:    SUB   R2,R0       ;SUBTRACT MULTIPLICAND FROM PARTIAL PRODUCT
        ASR   R0
        ROR   R1
        BR    JMPRET
```

В обоих приведенных выше алгоритмах для предотвращения переполнения величина m должна быть меньше, чем $(2 ** 15 - 1)/1{,}5$. Хотя среднее число нулей или единиц в n-битовом слове равно $n/2$, можно показать, что число переходов от нуля к единице или наоборот равно $n/3$. Рассмотренная выше программа работает медленнее, чем программы, основанные на традиционных методах умножения, но она взята в качестве примера в связи с тем, что обладает рядом преимуществ. Во-первых, реализация алгоритма интересна с точки зрения применения таб-

лицы переходов и, во-вторых, микропрограммная реализация алгоритма Бута дает существенное повышение скорости выполнения умножения с фиксированной точкой. Наконец, этот алгоритм служит для достижения высокой скорости умножения в некоторых современных микросхемах-умножителях.

Расширенный набор команд (EIS) включает команду умножения дополнений до 2, имеющую следующий синтаксис:

```
MUL SRC,Rn
```

Операнд SRC может быть задан любым из восьми методов адресации системы PDP-11. Если в качестве Rn используют R0, R2 или R4, то результат будет содержаться в парах регистров (R0 и R1), (R2 и R3) или (R4 и R5) соответственно, а при задании R1, R3 или R5 в этих регистрах хранятся только младшие 16 бит результата. Последний прием полезен для индексных вычислений, а умножение 16 бит на 16 бит с получением 32-битового результата целесообразно, например, для цифровой обработки сигналов. Допустим, что реальная и мнимая части последовательности комплексных чисел хранятся соответственно в четных и нечетных элементах массива. Тогда сумма квадратов частей может быть эффективно вычислена и запомнена в исходном массиве следующим образом:

```
        MOV     #A,R0           ;ADDRESS OF 1'ST ELEMENT OF ARRAY
        MOV     R0,R1           ;DUPLICATE
        MOV     #n,-(SP)        ;NUMBER OF COMPLEX DATA POINTS
        ;
LOOP:   MOV     (R0)+,R2        ;Re[A( )]
        MUL     R2,R2           ;Re[A( )]**2
        MOV     (R0)+,R4        ;Im[A( )]
        MUL     R4,R4           ;Im[A( )]**2
        ADD     R5,R3           ;32-BIT ADDITION
        ADC     R2
        ADD     R4,R2           ;Re[A( )]**2 + Im[A( )]**2
        MOV     R2,(R1)+        ;STORE HIGH WORD
        MOV     R3,(R1)+        ;STORE LOW WORD
        DEC     (SP)
        BNE     LOOP
        ;
        TST     (SP)+           ;POP COUNT
```

Заметим, что поскольку в «MUL SRC, Rn» n четное, то результат Rn * SRC помещается в Rn и Rn + 1. Приведенный прием возведения в степень корректен и эффективен по времени.

В цифровой обработке сигналов от аналого-цифровых преобразователей получают «целые» числа, и их часто приходится умножать на такие дробные, как синус или косинус. Дроби могут умножаться на некоторую степень 2 так, чтобы целая часть результата была меньше, чем 2 ** 15 — 1, но больше, чем 2 ** 14, и хранилась в форме таблицы целых значений. Допустим, что должно быть вычислено следующее выражение:

```
x(i) ← ,x(i)*[0.54-0.46*cos(2π[i-1]/N)], ⌊i = 1, 2, ..., N
```

где $x(i)$ — значения, представленные в виде дополнения до 2. Для определения множителя приведенное ниже выражение может быть запрограммировано на Фортране.

```
NW(I)=(2.**15-1.)*(0.54-0.46*COS(2.*PI*XI/XN))
```

где XI = I — 1 и XN = N, так что в этом выражении |NW(I)| ⩽ ⩽ 2 ** 15 — 1. Далее, если $x(i)$ находится так, как показано ниже,

```
        MOV     #X,R0           ;I/O ARRAY STARTING ADDRESS
        MOV     #NW,R1          ;WEIGHTING ARRAY STARTING ADDRESS
        MOV     #n,R2           ;n = NUMBER OF POINTS IN ARRAYS
        ;
LOOP:   MOV     (R0),R4         ;R4     <- x(i)
        MUL     (R1)+,R4        ;R4,R5 <- x(i)*NW(I)
        MOV     R4,(R0)+        ;HIGH WORD = PRODUCT/2**16
        SOB     R2,LOOP         ;DECREMENT R2, BRANCH IF NOT ZERO
```

то результат может быть интерпретирован следующим образом:

```
x(i) <- [x(i)*[(2.**15-1.)*[0.54-0.46*cos(2π[i-1]/N)]]]/[2.**16]

     = 0.5*x(i)*[0.54-0.46*cos(2π[i-1]/N)],
```

Поскольку оказываются использованными 16 бит старшего слова результата, занимающего 32 разряда, то он должен быть разделен на 2 ** 16. Такой способ интерпретации удобен и эффективен по времени при цифровой обработке сигналов, в частности при выполнении быстрого преобразования Фурье.

5.2.5. Деление

Деление сложнее умножения по нескольким причинам.

1. Результатом деления являются два числа — частное и остаток. Частное от деления целого числа на целое редко бывает целым, и правильность результата определяется как частным, так и остатком.

2. Максимальное значение, которое может быть получено при выполнении деления, равно бесконечности. В некоторых случаях значение частного не умещается в заданное число разрядов результата и в итоге происходит переполнение.

Частное от деления можно представить как количество вычитаний делителя из делимого до тех пор, пока делимое сохраняет знак. Это дает возможность организации деления путем повторения последовательности вычитаний с проверкой знака делимого. Если произошла перемена знака, то нужно уменьшить счетчик числа вычитаний на единицу, а делимое и делитель сложить. Теперь делимое будет равно остатку, а счетчик числа вычитаний равен частному.

Быстрее работают алгоритмы деления без восстановления, в них из делимого *вычитается* делитель, умноженный на степень двойки. Если знак результата, т. е. остатка, не изменился, то нужно прибавить ту же степень двойки к частному, в противном случае частное не увеличивается. Так повторяется для уменьшаю-

щихся степеней двойки до тех пор, пока не будет достигнута степень, равная нулю.

Когда же знак остатка меняется, вычитание заменяется *сложением*, причем слагаемыми служат произведение степени 2 на делитель и остаток. Сложение осуществляется до нового изменения знака, после чего значение степени складывается с частным Для иллюстрации приведем пример:

```
ВЫЧИСЛИТЬ  17/5 = 3, + 2  ОСТАТОК

                                    Q = 0
  17 - 5 * 8 = -23;                 Q = Q + 0 = 0
 -23 + 5 * 4 =  -3;                 Q = Q + 0 = 0
  -3 + 5 * 2 =  +7;                 Q = Q + 2 = 2
   7 - 5 * 1 =  +2;                 Q = Q + 1 = 3
```

В приведенной ниже программе для умножения 16-разрядных целых использован алгоритм без восстановления.

```
ONE=%0
DIVS=%1
;NON-RESTORING DIVISION PROGRAM
;DIVIDEND, DIVISOR ASSUMED 16-BIT POSITIVE INTEGERS PASSED VIA R5
DIVNRS:
        MOV     #1,R0           ;USED FOR QUOTIENT BIT SETTING
        CLR     @10.(R5)        ;OVERFLOW FLAG INITIALLY CLEARED
        CLR     R2              ;CLEAR REMAINDER
        MOV     @2(R5),R3       ;DIVIDEND IN R3
        BEQ     ZERO            ;ZERO DIVIDEND?
        MOV     #16.,R4         ;LOOP COUNT
        MOV     @4(R5),DIVS     ;DIVISOR  IN R1
        BEQ     OVFL            ;DIVISOR = 0, NO GO!
POSSTP:
        ASL     R3              ;LEFT SHIFT R2,R3
        ROL     R2
        SUB     DIVS,R2         ;SUBTRACT DIVISOR FROM REMAINDER
        BMI     NEG             ;BRANCH IF SIGN CHANGE
        BIS     ONE,R3          ;NO, SET QUOTIENT BIT
        SOB     R4,POSSTP       ;AND AROUND AGAIN
        BR      FIN
NEGSTP:
        ASL     R3              ;LEFT SHIFT R2,R3
        ROL     R2
        ADD     DIVS,R2         ;ADD DIVISOR TO REMAINDER
        BMI     NEG             ;DO AGAIN IF SIGN STILL NEGATIVE
        BIS     ONE,R3          ;NO, SET QUOTIENT BIT
        SOB     R4,POSSTP       ; AND BACK TO SUBTRACTING DIVISOR
        BR      FIN
NEG:
        SOB     R4,NEGSTP
FIN:
        TST     R2              ;REMAINDER NEGATIVE
        BGE     OK              ;NO
        ADD     DIVS,R2         ;YES, ADD DIVISOR TO REMAINDER
OK:     MOV     R2,@6(R5)       ;REMAINDER
        MOV     R3,@8.(R5)      ;QUOTIENT
        RTS     PC
OVFL:
        INC     @10.(R5)        ;SET OVERFLOW FLAG FOR ERROR
        RTS     PC
ZERO:
        MOV     @4(R5),@6(R5)   ;REMAINDER IS DIVISOR IF DIVIDEND ZERO
        CLR     @8.(R5)         ;QUOTIENT IS ZERO
        RTS     PC
        .END
```

В расширенном наборе EIS команда деления имеет синтаксис

DIV SRC, Rn,

где в качестве Rn должны быть R0, R2 или R4. Длинное 32-разрядное делимое находится в Rn и Rn + 1; оно делится на SRC.

после чего частное помещается в Rn, в остаток — в Rn + 1. С помощью этой команды деление одного 16-разрядного числа на другое может быть запрограммировано так:

```
MOV  DIVDND,R3    ; 16-разрядное делимое
SXT  R2           ; расширение знака для формирования 32-битового делимого
DIV  DIVSOR,R2    ; разделить
MOV  R2,QUOT      ; частное
MOV  R3,REM       ; остаток
```

В параграфе 6.8.1 приведен пример, где команда MUL используется для 32-разрядных чисел.

5.2.6. Расширенный набор команд перехода

В более сложных командах условного перехода, как и в командах арифметики с повышенной точностью и сдвиговых командах, употребляются биты переноса и переполнения. В некоторых случаях рассматриваются только абсолютные значения чисел, например в инструкциях BEQ и BNE, в других 15-й бит числа интерпретируется как знак, в частности в BPL и BMI. Читателю могут оказаться полезными и такие команды, как переход по «меньше» BLT, переход по «меньше или равно» BLE, переход по «больше» BGT и переход по «больше или равно» BGE. В этих командах используют биты C и V, а также N и Z. Теперь мы объясним, как они работают.

Беззнаковые команды условного перехода BHI, BLOS, BHIS и BLO работают с 16-разрядными числами, оперируя ими как арифметическими последовательностями вида:

наибольшее значение	177777
	177776
	.
	.
	.
	000001
наименьшее значение	000000

Но, с другой стороны, такие последовательности в знаковой форме дополнения до двух представляются так:

наибольшее положительное значение	077777
	077776
	.
	.
	000001
наименьшее положительное значение	000000
наименьшее отрицательное значение	177777
	177776
	.
	.
наибольшее отрицательное значение	100000

Различие между двумя типами последовательностей очевидно, однако пока не ясно, почему они должны различаться в командах условного перехода. Простой пример позволяет пояснить это

Допустим, что необходимо проанализировать результаты выполнения команды, операндами которой служат 16-разрядные значения со знаком в форме дополнения до 2. Тогда при сравнении отрицательного значения источника —32767 с положительным значением приемника 32767 значения битов условий по результату вычитания (SRC) — (DST) будут:

```
  -32767.        100001        100001    C    Z = 0
-  32767.  →  - 077777  →  + 100001          N = 0
 --------      --------      --------          C = 0
       2.        000002      1)000002          V = 1
```

Значения бит C и V будут установлены в соответствии с определением команды вычитания SUB:

1. ПЕРЕНОС равен содержимому MSB.

2. ПЕРЕПОЛНЕНИЕ формируется тогда, когда операнды имеют противоположный знак, а знак получаемого результата совпадает со знаком вычитаемого, т. е. операнда источника в команде SUB и операнда приемника в команде CMP. Следовательно, последовательность двух инструкций

```
CMP     #-77777,#77777
BMI     LESS
```

не вызывает перехода, поскольку N = 0.

С другой стороны, следующие две команды

```
CMP     #-77777,#77777
BLT     LESS
```

приведут к выполнению перехода. Различие между этими командами условного перехода заключается в том, что они по-разному интерпретируют биты условий. Простые команды перехода просто анализируют знаковый разряд, в то же время более сложные, кроме того, анализируют биты переноса и переполнения, которым присваивается значение в результате выполнения операции сравнения. Используя *сложение по модулю два* (∀) и *логическое ИЛИ* (∨), работу команд можно описать так:

Команда	Символическая форма	Функция. Переход, если
Переход по «меньше»	BLT X	N ∀ V = 1
Переход по «меньше или равно»	BLE X	Z ∨ (N∀V) = 1
Переход по «больше»	BGT X	Z ∨ (N∀V) = 0
Переход по «больше или равно»	BGE X	N ∀ V = 0

В дополнение к уже описанным командам условного перехода имеется еще несколько *не связанных со знаком числа* простых команд, которые могут оказаться полезными (они приводятся ниже):

Команда	Символическая форма	Функция. Переход, если
Переход, если выше	BHI X	C = 0 и Z = 0
Переход, если ниже или то же	BLOS X	CVZ = 1
Переход, если выше или то же	BHIS X или BCC X	C = 0
Переход, если ниже	BLO X или BCS X	C = 1
Переход, если бит переполнения = 0	BVC X	V = 0
Переход, если бит переполнения = 1	BVS X	V = 1

С помощью первых четырех из них программист может анализировать все арифметические результаты, представленные в форме дополнения до двух. Последние шесть команд дают возможность анализировать как беззнаковые арифметические результаты, так и отдельные биты условий. Например, команда CMP *заносит ноль в бит* C, если при вычислении разности X — Y происходит перенос, и *единицу в бит* C в противном случае. Пусть Y = X — K, где X, Y и K — *беззнаковые* двоичные числа; тогда, используя операцию дополнения до единицы, обозначаемую знаком «~», можно записать:

```
X - Y  =  X + ~(X - K)      + 1
       =  X + ~(X + ~K + 1) + 1
       =  (X + 1) + ~(X + 1) + K
       =  177777 + K
```

Если K > 0, т. е. X > Y, то *будет* выработан перенос и поэтому C = 0; в таком случае обе команды BHI и BHIS работают верно. Поскольку в PDP-11 адреса представляются *беззнаковыми двоичными числами*, для сравнения адресов можно воспользоваться командой CMP и беззнаковыми командами переходов.

5.3. ЛОГИЧЕСКИЕ ОПЕРАЦИИ

Команды, реализующие логические операции, относятся к группе двухоперандных (см. параграф 3.3.3). Сюда входят команды BIS, BIC и BIT, которые служат для установки, очистки и анализа состояния бит в слове. Ниже приводится их определение, в нем для обозначения логического отрицания употребляется символ «~», а для обозначения логического И — символ «∧».

Команда	Символическая форма		Функция
Установить бит в 1	BIS	X,Y	(Y) ← (X) ∨ (Y) (ИЛИ OR)
Обнулить бит	BIC	X,Y	(Y) ← ~ (X) ∨ (Y)
Проверить бит	BIT	X,Y	Вычислить (X) ∧ (Y), установить коды условий
Сложение по модулю	XOR	Rn,Y	(Y) ← (Rn) ∀ (Y)

В машинах 11/04, 05 и 10 команда XOR отсутствует. В целом эта группа инструкций следующим образом устанавливают коды условий в слове состояния процессора:

1) биты N и Z устанавливаются в зависимости от условий;
2) бит V обнуляется;
3) бит C не изменяется.

В системе PDP-11 все устройства ввода-вывода имеют управляющие регистры и регистры состояния, к которым можно адресоваться как к ячейкам памяти. У каждого из них имеется собственный адрес, и его состояние можно анализировать или изменять при помощи логических команд. Если необходимо проверить содержимое разрядов 15 и 7 в регистре состояния STAT, можно написать:

```
.
BIT     #100200,STAT    ;TEST BITS 15 AND 7
BMI     BIT15           ;BIT 15 IS SET
BNE     BIT7            ;BIT 7 IS SET
:
```

При удалении из этого фрагмента команды BMI, BNE будет проверять состояние обоих разрядов — и 15, и 7.

5.4. АРИФМЕТИКА С ПЛАВАЮЩЕЙ ТОЧКОЙ

При выполнении операций с плавающей точкой используются знаки, порядки и мантиссы операндов с плавающей точкой. При выполнении операций с фиксированной точкой умножение n-разрядного операнда требует n-кратного выполнения внутреннего цикла, включающего сдвиг и сложение в зависимости от условий, в случае если умножение реализовано программно. Однако при работе с плавающей точкой не только умножение, но и сложение реализуется достаточно сложными алгоритмами.

5.4.1. Умножение и деление

Результатом операции над двумя числами с плавающей точкой

$$X_1 = s_1 * f_1 * 2^{**}(e_1)$$

и

$$X_2 = s_2 * f_2 * 2^{**}(e_2)$$

является

$$P = (s_1 * s_2) * (f_1 * f_2) * 2^{**}(e_1 + e_2)$$

$$= s_p * f_p * 2^{**}(e_p)$$

Знак результата определяется путем сложения по модулю 2 знаков сомножителей. Поскольку мантиссы нормализованы, для нормализации результирующей мантиссы, возможно, потребуется сдвинуть ее на один разряд, т. е.

$$0.5 \le f_1 < 1\ ,\ 0.5 \le f_2 < 1 \text{ so that } 0.25 \le f_p < 1.$$

Иными словами, если результирующая мантисса меньше, чем 0,5, то она умножается на 2, а порядок уменьшается на единицу. Очевидно, что основное место при выполнении умножения чисел с плавающей точкой занимает *умножение мантисс.* В системе PDP-11 перемножение двух чисел одинарной точности с плавающей точкой требует умножения двух 24-разрядных беззнаковых целых. Допустим, что аппаратные возможности (например, EIS) или возможности программного обеспечения позволяют умножать 16-разрядные числа; в таком случае умножение 24-разрядных чисел может быть представлено так:

$$f_1 * f_2 = (f_1' + f_1'' * 2^{-16}) * (f_2' + f_2'' * 2^{-16})$$

$$= f_1' * f_2' + (f_1' * f_2'' + f_2' * f_1'') * 2^{-16}$$

где f' и f'' — соответственно беззнаковые 16- и 8-разрядные целые. Деление с плавающей точкой

$$Q = X_1 / X_2 = (s_1 * s_2) * (f_1 / f_2)\ 2^{**}(e_1 - e_2)$$

Очевидно, что и здесь потребуется выполнить нормализацию порядка.

5.4.2. Сложение и вычитание

При сложении или вычитании чисел с плавающей точкой основную сложность составляют не собственно операции сложения или вычитания. В связи с тем, что в них используются операнды с нормализованными мантиссами, прежде всего необходимо *выравнивание порядков* операндов. Для этого операнд с меньшим значением порядка сдвигается вправо и соответственно его порядок увеличивается. Только после этого выполняется сложение или вычитание. Наконец, дробная часть результата должна быть заново нормализована; при этом может потребоваться сдвиг на всю ее длину.

В целом можно сказать, что если при умножении и делении с плавающей точкой доминировало *умножение* или *деление* дробной части, то при сложении и вычитании основное место занимают *нормализации* до и после соответствующей операции.

Обычно пользователям PDP-11 не приходится уделять большое внимание подробностям операций с плавающей точкой, даже в том случае, если отсутствуют требуемые аппаратные или программные средства, потому что при использовании языков высокого уровня и при программировании на языке Ассемблера программист может получить доступ к готовым пакетам соответствующих программ. Однако подобно тому, как при переходе

к программированию на языке Ассемблера с применением различных методов адресации PDP-11 повышается эффективность работы с данными различных типов, с помощью команд с плавающей точкой повышается эффективность функционирования всего комплекса.

5.4.3. Набор команд с плавающей точкой (FIS)

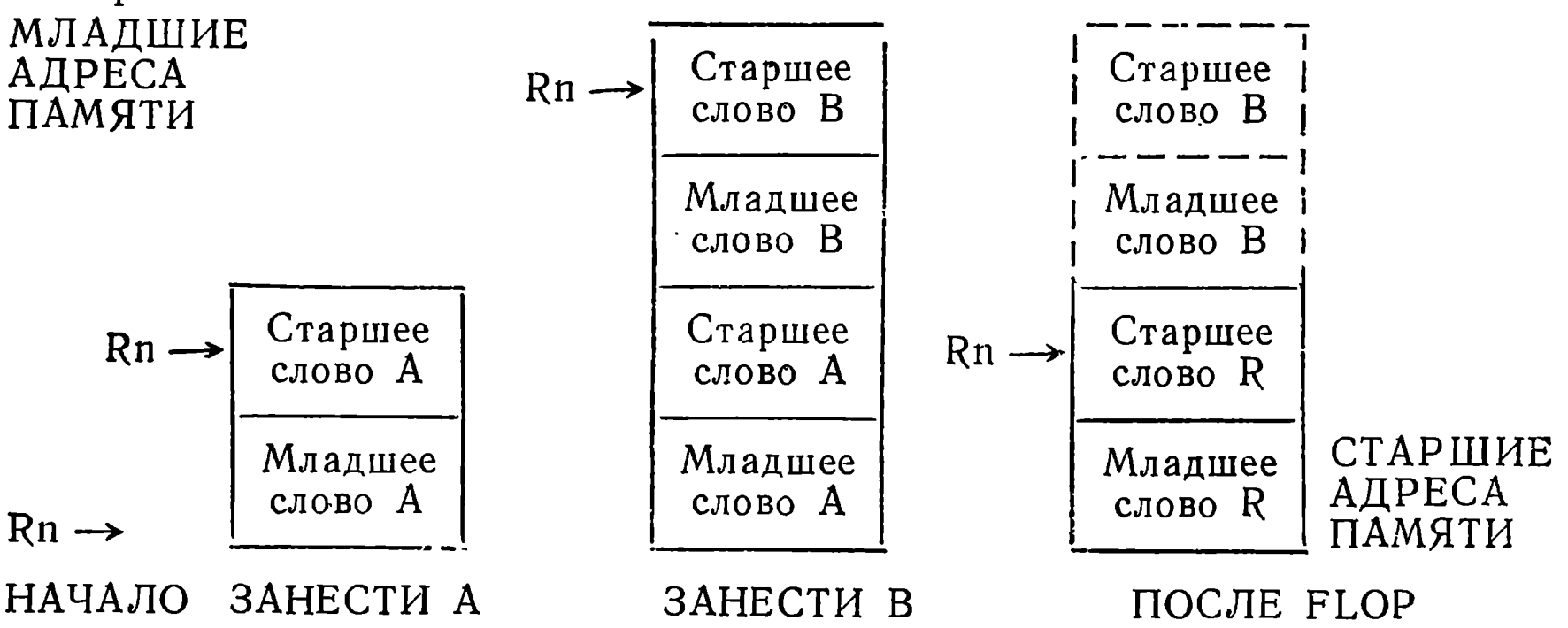
Набор FIS состоит из четырех команд — FADD, FSUB, FMUL и FDIV — и является дополнительным средством для машин PDP-11/35, 40 и LSI-11(PDP-11/03). Эти команды *выбирают* операнды из соответствующего стека, вычисляют результат и заносят его в стек. Схематически выполнение операции с плавающей точкой FLOP можно представить следующим образом:

$$r = a \text{ «FLOP» } b,$$

где a хранится в A и A + 2, b — в B и B + 2, а результат r — в R и R + 2. Она может быть описана такой последовательностью команд:

```
        MOV #C,Rn         ;INITIALIZE STACK
PUSH:   MOV A+2,-(Rn)     ;PUSH LOW  A
        MOV A  ,-(Rn)     ;PUSH HIGH A
        MOV B+2,-(Rn)     :PUSH LOW  B
        MOV B  ,-(Rn)     ;PUSH HIGH B, CONTENTS Rn=#C-8
OPR:    FLOP Rn
                          ;CONTENTS Rn=#C-4
POP:    MOV (Rn)+,R       ;POP HIGH R
        MOV (Rn)+,R+2     ;POP LOW  R,  CONTENTS Rn=#C
```

Результатом работы будет $a + b$, $a - b$, $a * b$ или a/b, в зависимости от того, заменим ли мы FLOP на FADD, FSUB, FMUL или FDIV соответственно. Для иллюстрации того, что происходит в стеке при выполнении этих команд, приведена следующая диаграмма:



Заметим, что по окончании выполнения программы операнд B физически остается в стеке, но не адресуется как элемент структуры данных. Ранее было показано, что количество операций можно

уменьшить, если воспользоваться сохранившимся в стеке операндом. Однако при этом нельзя употреблять системный стек, поскольку его содержимое может быть изменено без ведома пользователя (см. гл. 7).

Вычисление факториала. Факториал имеет смысл только для целых чисел, но с помощью FIS можно вычислить «целый» 24-разрядный результат. В примере подпрограмма «FACT», вызываемая по команде JSR R5, FACT, итеративно вычисляет $N!$, т. е. $N! = N(N-1)\ (N-2) \ldots 1$

```
        JSR R5,FACT
        .WORD n                 ;N INPUT, HIGH RESULT OUT
        .WORD 0                 ;LOW RESULT OUT
          .
          .
          .
FACT:
        MOV (R5),R2             ;R2 ← N, N > 1
        DEC R2                  ;R2 ← N-1
        MOV ONE,RESULT          ;INITIALIZE RESULT
        MOV ONE+2,RESULT+2
        MOV ONE,X               ;INTIALIZE X
        MOV ONE+2,X+2
LOOP:
        MOV #ONE,R0             ;SET UP STACK
        FADD R0                 ;X ←  X+1
        FMUL R0                 ;RESULT ← RESULT*X
        SOB R2,LOOP

        .
        .
        .
        MOV (R0)+,(R5)+         ;RESULT FOLLOWS
        MOV (R0)+,(R5)+         ;SUB CALL
        RTS R5
;
ONE:    .FLT2 1.
X:      .FLT2 1.
RESULT: .FLT2 1.
```

.FLT2 — директива, которая превращает аргумент в эквивалентное 32-разрядное двоичное представление числа с плавающей точкой.

Вычисление многочлена. В системе VAX-11 продолжена тенденция реализации в виде машинных команд некоторых полезных алгоритмов. Например, команда POLY имеет такой синтаксис:

```
POLY    X,#n,PTABLE
```

где X — идентификатор числа с плавающей точкой, n — целое, PTABLE — идентификатор таблицы, содержащей n чисел с плавающей точкой $C(n)$, $C(n-1)$, ..., $C(0)$. На Фортране команда POLY может быть реализована так, как показано ниже, в предположении, что допустимо нулевое значение индекса элемента массива.

```
R = C(N)
DO 1 I =1,N
R = R * X + C(N-I)
```

Таким образом, POLY вычисляет выражение

$$R = C(0) + X*C(1) + X^{2}*C(2) + \ldots + X^{n-1}*C(n-1) + X^{n}*C(n)$$

На Ассемблере это можно реализовать так:

```
          JSR     R5,POLY
          .WORD   X
          .WORD   n
          .WORD   ENDTAB
                  .
                  .
PTABLE:   C(0) (high word)
          C(0) (low word)
            .
            .
          C(n) (high word)
          C(n) (low word)
ENDTAB:   .WORD   0
          .BLKW   9.
STACK:    .WORD   0
X:        .FLT2   x                    ;X


;
POLY:     MOV     (R5)+,R1             ;ADDRESS OF X HIGH
          MOV     (R1)+,R0             ;X HIGH
          MOV     (R1)+,R1             ;X LOW
          MOV     (R5)+,R2             ; n = ORDER OF POLYNOMIAL
          MOV     (R5)+,R3             ;ENDTAB = TABLE END+2
          MOV     #STACK,R4            ;SET UP USER STACK
;
          MOV     -(R3),-(R4)          ;C(n) LOW  = R LOW
          MOV     -(R3),-(R4)          ;C(n) HIGH = R HIGH
;
LOOP:     MOV     R1,-(R4)             ;X LOW
          MOV     R0,-(R4)             ;X HIGH
          FMUL    R4                   ;R ← R * X
          MOV     -(R3),-(R4)          ;C(n-I) LOW
          MOV     -(R3),-(R4)          ;C(n-I) HIGH
          FADD    R4                   ;R ← R + C(n-I)
          SOB     R2,LOOP
;
          MOV     (R4)+,R0             ;R HIGH
          MOV     (R4)+,R1             ;R LOW
          RTS     R5
```

Команда POLY особенно полезна для решения тригонометрических полиномиальных уравнений.

5.4.4. Процессор с плавающей точкой (FPP), его архитектура и система команд

Процессор с плавающей точкой микропрограммно реализован в модели PDP-11/60; как средство усиления арифметических возможностей он доступен на системах PDP-11/34, 45, 55, 60 и 70.

FPP состоит из шести 64-разрядных аккумуляторов и имеет мощную систему команд для работы с числами с плавающей точкой одинарной и двойной точности, а также для преобразования их в целые слова или слова увеличенной длины и обратно. На рис. 5.1 показана структура FPP и его связь с CPU. Команды FPP разделены на одно- и двухоперандные группы.

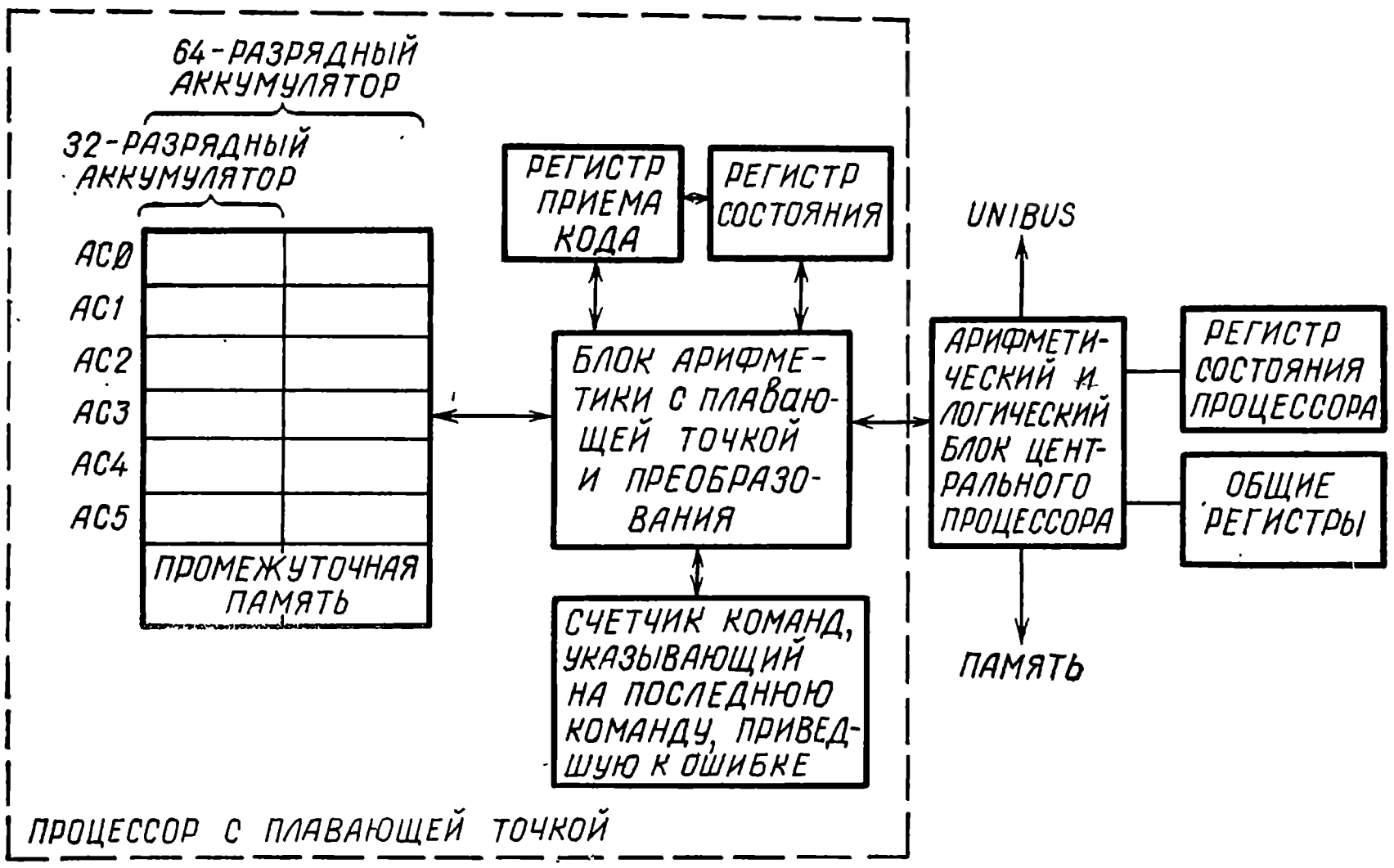
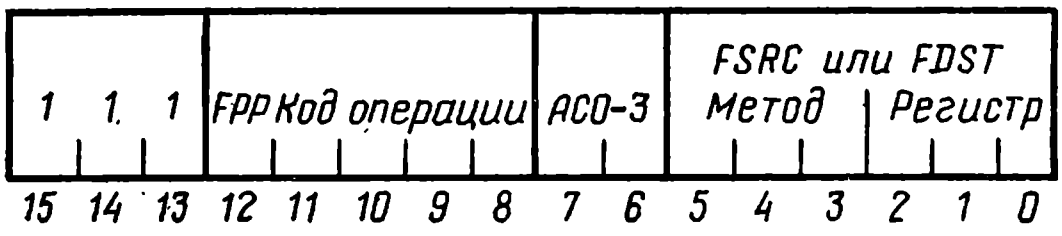


Рис. 5.1. Процессор с плавающей точкой (FPP) и CPU

Двухоперандные команды FPP. Эти команды имеют следующий формат:



В командах с плавающей точкой адрес источника SRC и адрес приемника DST задаются с использованием уже известных методов адресации PDP-11 с *двумя исключениями:*

1) в методе 0 биты регистра указывают на один из регистров AC0 — AC5;

2) в методах 2 и 4 адрес уменьшается или увеличивается на 4 или 8, в зависимости от того, в каком режиме находится FPP — с одинарной или двойной точностью.

Вторым операндом может быть только один из регистров AC0 — AC3. Следовательно, невозможна прямая связь между памятью и регистрами AC4 — AC5.

Режим работы FPP определяется доступным для пользователя битом в регистре состояния FPP. В отличие от команд с фиксированной точкой в PDP-11, где MSB определяет их ориентацию на слово или байт, команды FPP работают с числом с плавающей точкой одинарной или двойной точности в зависимости от значения бита в регистре состояния FPP.

Двухоперандные команды FPP делятся на *загружающие* и *сохраняющие*. Последняя буква мнемонического обозначения команды определяет одинарную или двойную точность.

Класс загружающих команд

Символический код	*Функция*
LDF,LDD	(AC) ← (FSRC)
ADDF,ADDD	(AC) ← (AC) + (FSRC)
SUBF,SUBD	(AC) ← (AC) - (FSRC)
MULF,MULD	(AC) ← (AC) * (FSRC)
DIVF,DIVD	(AC) ← (AC) / (FSRC)
CMPF,CMPD	*вычислить* (FSRC) - (AC), *установить* FPP *коды условий*
MODF,MODD	*вычислить* PROD = (FSRC) * (ACi), *тогда* (ACi V 1) ← N = INT[PROD] *и* (ACi) ← g = PROD - N. *Таким образом, если i=1 или 3, только g сохраняется в* (ACi).

Класс запоминающих команд

Символический код	*Функция*
STF,STD	(FDST) ← (AC)

Остальные команды обоих классов позволяют совместить загрузку или сохранение с преобразованием *из* любого режима с плавающей точкой в любой другой режим с плавающей точкой или *в* один из способов представления целых, и наоборот. Имеются также команды для работы со знаком и порядком, для перевода FPP в режим с плавающей точкой или для работы с целыми, а также для операций с регистром состояния FPP.

Помимо той многосторонности, которую обеспечивает развитая система команд с плавающей точкой, и возможности в некоторой степени совмещать во времени работу FPP с CPU на моделях 11/55, 70, а при наличии соответствующего расширения — и на 60, процессор FPP совместно с CPU образует машину с плавающей точкой и регистрами общего назначения, обладающую всеми преимуществами таких ЭВМ. Ниже приведены некоторые примеры.

Подпрограмма POLY с FPP.

```
          JSR     R5,POLYX
          .WORD   XX
          .WORD   n
          .WORD   ENDTAB
                  .
PTABLE:   C(0) (high word)
          C(0) (low word)
            .
            .
          C(n) (high word)
          C(n) (low word)
ENDTAB:   .WORD   0
XX:       .FLT2   x                    ;X
X=%0                                   ;DEFINE X AND R AS FP REGISTERS
R=%1
POLYX:    SETF                         ;SINGLE PRECISION MODE
          LDF     @(R5)+,X             ;X
          MOV     (R5)+,R2             ;n = ORDER OF POLYNOMIAL
          MOV     (R5)+,R3             ;ENDTAB = TABLE END+2
                                       ;NOTE: AUTODECREMENT IS BY 4
          LDF     -(R3),R              ;R ← C(n)
LOOP:
          MULF    X,R                  ;R ← R * X
          ADDF    -(R3),R              ;R ← R + C(n-I)
          SOB     R2,LOOP
;
          STF     R,ANS                ;ANSWER INTO MEMORY
          RTS     R5
ANS:      .FLT2   0
```

Использование регистров с плавающей точкой для временного хранения данных уменьшает потребность в памяти на вычисление по сравнению с применением FIS.

Умножение комплексных чисел. Комплексное умножение обычно требует четырех умножений и двух сложений, т. е. $(a + jb) * (c - jd) = (ac + bd) + j(bc - ad)$. Если c и d — константы, а вычисленные значения $c - d$, $c + d$ и d хранятся в табличном виде, то с помощью регистров FPP комплексное умножение можно выполнить тремя операциями умножения и тремя сложения.

```
SETF                                   ;SINGLE PRECISION MODE
MOV       #AB,R0                       ;POINTER TO a,
MOV       #CD,R1                       ;POINTER TO c - d, c + d, d
LDF       (R0)+,AC0                    ;AC0 ← a
LDF       (R0)+,AC1                    ;AC1 ← b
LDF       AC0,AC2                      ;AC2 ← a
ADDF      AC1,AC2                      ;AC2 ← a + b
MULF      (R1)+,AC0                    ;AC0 ← a*(c - d)
MULF      (R1)+,AC1                    ;AC1 ← b*(c + d)
MULF      (R1)+,AC2                    ;AC2 ← d*(a + b)
ADDF      AC2,AC0                      ;AC0 ← ac + bd
SUBF      AC2,AC1                      ;AC1 ← bc - ad
```

5.5. ВНУТРЕННЕЕ И ВНЕШНЕЕ ПРЕДСТАВЛЕНИЕ ДАННЫХ

Теперь рассмотрим, как представляются в памяти числа с фиксированной и с плавающей точкой и как они используются CPU при выполнении арифметических операций. Это важно для пони-

мания того, как числа вводятся в вычислительную систему и выводятся из нее.

В Фортране преобразование данных из *внешней* во *внутреннюю* форму определяется оператором FORMAT, связанным с оператором READ. Оператор FORMAT, указанный в операторе WRITE, служит для обратного преобразования. Однако вне зависимости от того, использует программист язык высокого уровня или нет, необходимость преобразования числа из внешней формы во внутреннюю и наоборот существует всегда.

Задача может быть поставлена точнее в терминах наиболее типичной для мини-ЭВМ операции ввода-вывода — работе на терминале. Если пользователь нажал клавиши, допустим, «3» и «5», то он может подумать, что сохранил в некоторой ячейке памяти восьмеричное число «35». Однако при более внимательном рассмотрении выясняется, что он всего лишь сохранил восьмеричные значения 63 и 65 в двух байтах. Очевидно, что необходимо выполнить преобразование генерируемого терминалом кода ASCII во внутреннее значение $35_{(8)}$. Если символ ASCII помещен в R0, а результат нужен в R1, то для преобразования можно использовать следующую подпрограмму:

```
CONV:   BIC.    #177770,R0      ;CLEAR ALL BUT LOWER 3 BITS
        ASH     #3,R1           ;PREVIOUS NUMBER * 8
        BIS     R0,R1           ;"OR" IN R0
        RTS     PC
```

Регистр R1 должен быть обнулен прежде, чем преобразуется первый символ.

При выводе на терминал решается обратная задача. В этом случае внутреннее представление должно быть преобразовано в код ASCII. Подпрограмма, которая читает 16-битовые слова в R1 и переписывает шесть символов, хранящихся в упакованном виде, в коде ASCII в буфер, к которому можно адресоваться по R2, представлена ниже

```
OUT:    MOV     #5,R0           ;LOOP COUNT
LOOP:   MOV     R1,R3           ;COPY WORD
        BIC     #177770,R3      ;ONE OCTAL VALUE
        ADD     #'0,R3          ;CONVERT TO ASCII
        MOVB    R3,-(R2)        ;STORE IN BUFFER
        ASH     #-3,R1          ;SHIFT RIGHT THREE
        SOB     R0,LOOP         ;DECR R0, BRANCH IF NOT ZERO

        BIC     #177776,R1.     ;GET LAST BIT
        ADD     #'0,R1          ;CONVERT TO ASCII
        MOVB    R1,-(R2)        ;STORE IN BUFFER
        RTS     PC.             ;DONE, RETURN
```

Использование «#'0» служит для формирования значения кода символа в ASCII путем прибавления к внутреннему представлению кода нуля.

В гл. 7 будет показано, каким образом организуется обмен между CPU и внешними устройствами.

5.6. ВВЕДЕНИЕ В СТРУКТУРЫ ДАННЫХ

Программист встречается со структурами данных в двух случаях. Во-первых, они играют ключевую роль в алгоритме и, следовательно, в проектировании программы; во-вторых, вносят ограничения и определяют процесс описания задачи и кодирования программы. Поскольку структуры данных представляют собой средство для представления информации, они имеют внешнюю и внутреннюю формы. Например, десятичное число, имеющее математический вид 0,465 * 10 ** 2, представляется внутри ЭВМ в формате с плавающей точкой 0101101011101000. Программист, работающий на Фортране, должен потратить немало времени для изучения различных спецификаций формата, чтобы научиться описывать преобразования из внешней формы во внутреннюю и наоборот.

Изучение структур данных является настолько фундаментальной частью изучения алгоритмов и программирования, что эта тема явно или неявно затрагивается во всех учебниках. Например, читатели этой книги уже знакомы с элементарными структурами данных, доступными при программировании на PDP-11, такими, как биты, байты, слова, регистры, а также с представлением и значением таких понятий, как адрес, указатель, символ и стек. Для широкого понимания информационных структур читателю были предложены структуры вычислительных машин и программ, включая преобразования символических информационных конструкций, образующих программу.

Теперь обсудим строение более сложных структур данных и работу с ними. Читателя не должно удивлять, что они базируются на уже хорошо известных элементарных конструкциях. Мы будем рассматривать массивы, стеки, очереди, кольцевые буфера, списки, таблицы идентификаторов и другие формы организации данных. Все они представляют собой не что иное, как комбинации элементарных типов данных, по-новому связывающие воедино биты, байты, слова и т. д.

Опытный программист внимательно относится к выбору форм представления данных и использует те из них, с которыми его программа могла бы работать эффективно. Выбор новой структуры данных должен быть тщательно обоснован, поскольку его применение может сильно повлиять на размеры и скорость исполнения программы. Рассмотрим некоторые такие структуры для того, чтобы программисту, который собирается программировать на языке Ассемблера, было понятно, что они представляют собой и как с ними работать. Естественно начать с *массива* — основного типа структур данных, известного каждому программисту, работающему на Фортране.

5.6.1. Массивы

Простейший вид массива — это *одномерная структура*, которая представляет собой набор однотипных элементов данных, образую-

щих блок, причем этот блок идентифицируется одним именем. Массив может быть легко отображен из внешней концептуальной формы во внутреннюю путем назначения соответствия между последовательными ячейками памяти и элементами массива. Преимущества использования последовательных ячеек очевидны; это позволяет употребить для доступа к элементам массива индексную или автоинкрементную адресацию (см. гл. 2).

Многомерные массивы — это тоже часто применяемые программистом, работающим на Фортране, структуры. Их элементы хранятся по столбцам в ячейках памяти в порядке возрастания адресов. В качестве примера на рис. 5.2 показано расположение

Элемент	Позиция	Элемент	Позиция
A(1,1,1)	0	A(1,2,1)	3
A(2,1,1)	1	A(2,2,1)	4
A(3,1,1)	2	A(3,2,1)	5

Элемент	Позиция	Элемент	Позиция
A(1,1,2)	6	A(1,2,2)	9
A(2,1,2)	7	A(2,2,2)	10
A(3,1,2)	8	A(3,2,2)	11

Рис. 5.2. Размещение в памяти массива A(3, 2, 2)

в памяти трехмерного массива A (3, 2, 2). Из рисунка ясно, что многомерный массив может быть представлен как набор одномерных, однако в этом случае для доступа к элементам не может служить простая индексация.

Адресация ячейки элемента трехмерного массива осуществляется с помощью *уравнения адресов,* описывающего отображение трехмерного массива в одномерную память ЭВМ. Если массив имеет размерности I, J и K, то адрес конкретного элемента A(i, j, k) вычисляется следующим образом:

$$\text{адрес}\,[A(i,j,k)] = \text{адрес}\,[A(1,1,1)] + (i-1) + I*(j-1) + I*J*(k-1)$$

Приведенное уравнение служит для преобразования значений трех индексов в один *линейный индекс* для адресации области памяти, занимаемой элементом массива. Пример, приведенный на рис. 5.2, дает возможность убедиться в том, что элемент A (2, 1, 2) будет занимать седьмую позицию относительно A(1, 1, 1).

В общем случае адресное уравнение для многомерного массива A(I, J, K, ..., M, N) имеет вид

$$\text{адрес}\,[A(i,j,k,\ldots,m,n)] = \text{адрес}\,[A(1,1,1,\ldots,1,1) + (i+1) + I*(J*(\ldots M*(n-1)+m-1)\ldots j-1)$$

Оно отражает тот факт, что по мере увеличения размерности возрастает сложность вычисления адреса. Даже для трехмерного массива вычисление результата требует выполнения трех операций умножения, и в малых ЭВМ, не имеющих соответствующего оборудования, эта задача требует значительного времени.

Упрощенное вычисление адреса массива. При определении адреса элемента массива следует учесть два момента. Во-первых, как уже было упомянуто, желательно выполнять меньше операций умножения. Во-вторых, в таких языках, как Фортран, вычисление адреса имеет динамический характер, оно откладывается до того момента, пока не потребуется необходимый элемент массива. На основании этих положений строится следующий часто применяемый прием. Для массива

```
ARRAY(K1,K2,...,Kn)
```

где n задает размерность, существует соответствующий *блок описания*, называемый иногда дополнительным вектором, который содержит информацию, представленную на рис. 5.3.

N слова	*Содержимое*
1	SIZE, N
2	N*K1
3	N*K1*K2
n	N*K1*...*Kn-1
n+1	address of first element of array

Рис. 5.3. Блок описания массива (ADB)

Здесь $SIZE = N * K1 * K2 * \ldots * Kn$ — число слов на один элемент массива; оно определяется типом данных и может быть описано следующим образом:

Описание массива	Тип	N
Целый	0	1
Действительный	1	2
Двойной точности	2	3
Комплексный	3	4
.	.	.

Информация, содержащаяся в блоке описания массива, становится доступной подпрограмме, которая в процессе выполнения вычисляет адрес первого слова элемента массива в том случае, когда элемент содержится в нескольких словах. При каждом вызове этой подпрограммы следующим образом вычисляется значение *адресного полинома*:

Дан массив ARRAY (K1, K2,..., Kn), тогда адрес элемента ARRAY (k1,k2,..., kn), где $1 \leqslant kj \leqslant Kj$, $j = 1,2,\ldots,n$, равен

$$A = WD(n+1) + (k1-1)*N + (k2-1)*WD(2) + \ldots + (kn-1)*WD(n),$$

где WD(2), WD(3) служат для обозначения слов 2, 3,... в блоке описания данного массива.

Обсудим преимущества предлагаемого приема. Во-первых, адресный полином заменяется более простой суммой произведений. Во-вторых, число элементов в новом уравнении возрастает линейно в соответствии с ростом размерности массива. В-третьих, он позволяет заранее, во время трансляции вычислить значения некоторых произведений, например N * K1, N * K1 * K2,..., не затрачивая на это время в процессе выполнения программы. В-четвертых, наличие обобщенной формы представления структуры данных в виде блока описания массива позволяет резервировать память динамически во время работы программы. Читатели, знакомые с Алголом, обнаружат в последнем пункте важный атрибут языка с блочной структурой, позволяющего использовать переменные значения размерностей массивов. На практике изменяемая размерность массива и динамическое резервирование места для него позволяют программисту более эффективно использовать ограниченный объем памяти небольшой вычислительной машины.

Пример. Допустим, что требуется на PDP-11 создать массив M(I, J, K) типа INTEGER, где I = 1, J = 5 и K = 7. На языке Ассемблера резервирование области и создание блока описания массива, показанного на рис. 5.3, будет выглядеть так:

```
        .
        .
N=1             ;ARRAY TYPE IS INTEGER
SIZE=N*3*5*7    ;SIZE OF ARRAY = N*I*J*K WORDS [n = 3]
M:      .BLKW SIZE      ;STORAGE RESERVATION FOR ARRAY
        .BYTE N,SIZE    ;ADB WD(1)
        3               ;ADB WD(2) = I
        15.             ;ADB WD(3) = I*J
ADB:    M               ;ABD WD(4), ADDRESS OF M(1,1,1)
        .
        .
```

Доступ к любому элементу, например M(2, 3, 4), может быть осуществлен с применением подпрограммы вычисления адресов элементов массива AEAS, ее вызов выполняется при помощи следующих команд:

```
        JSR     R0,AEAS ;CALL SUBROUTINE
        ADB             ;ADDRESS OF ADB WD(4)
        2.              ;FIRST SUBSCRIPT  = i
        3               ;SECOND SUBSCRIPT = j
        4               ;THIRD SUBSCRIPT  = k
        0               ;ARRAY ELEMENT ADDRESS WILL BE PUT HERE
NEXT:                   ;NEXT INSTRUCTION
```

Блок-схема программы AEAS для случаев, когда N = 1 или 2, показана на рис. 5.4. Если же сделать N параметром адресного

полинома, то появится возможность доступа к элементам массивов любого типа, нужно только добавить значение N в блок описания. На рис. 5.5 приведена подпрограмма вычисления адресов,

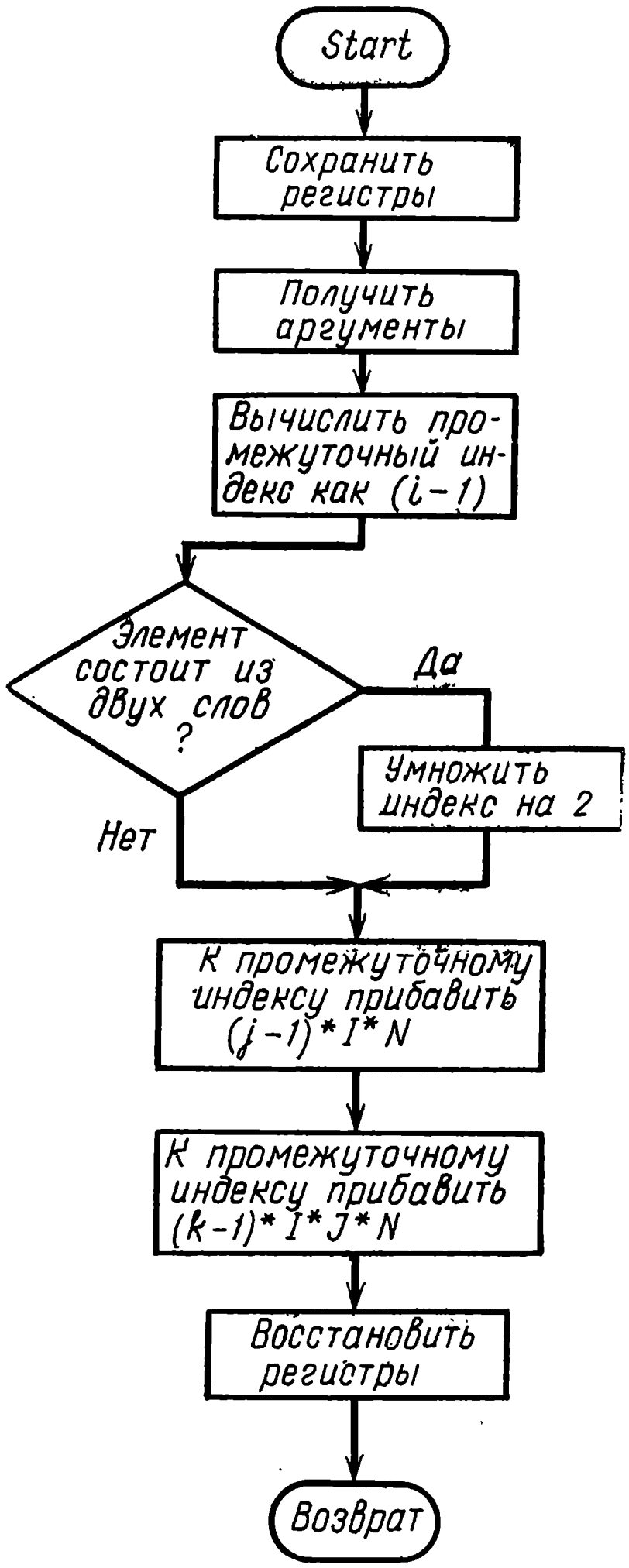


Рис. 5.4. Блок-схема вычисления адреса элемента массива

в ней используются подпрограммы для сохранения содержимого регистров SAVE и восстановления REST. Задав N равным 2, можно, например, зарезервировать память для массива действительных чисел XM(3, 5, 7), и в таком случае при вызове JSR R0, AEAS будет вычислен адрес элемента XM(2, 3, 4).

В этой подпрограмме адрес элемента $M(i, j, k)$, выраженный в байтах, вычисляется по формуле

```
M(i,j,k) = M(1,1,1) + [(i-1) + I*(j-1) + I*J*(k-1)]*2*N
               ^                 ^           ^ ^ ^
           word 4            word 2      word 3
           of ADB            of ADB      of ABD
```

где

```
    N = number of words per element = 1, 2, or 4
I,J,K = maximum values of i,j,k respectively.
```

Алгоритм требует нескольких операций умножения, но они заменяются командами сдвига влево и последовательностью команд сложения; этот прием позволяет избежать вызова подпрограмм умножения, если отсутствует команда MUL.

```
AEAS:   JSR     PC,SAVE         ;SAVE R1-R5
        CLR     R1              ;HOLDS RELATIVE DISPLACEMENT
        MOV     (R0)+,R2        ;ARRAY DESCRIPTOR BLOCK ADDRESS, ADB
;*******CALCULATE (i-1) + I*(j-1) + I*J*(k-1) ************************
        MOV     (R0)+,R3        ;FIRST SUBSCRIPT = i
        DEC     R3              ;  LESS ONE = i-1
        BLE     SKIP            ;IGNORE IF ≤ 0
        ADD     R3,R1           ;ADD TO RELATIVE DISP = i-1
SKIP:   MOV     (R0)+,R3        ;SECOND SUBSCRIPT = j
        DEC     R3              ;  LESS ONE = j-1
        BLE     SKIP1           ;IGNORE IF ≤ 0
        MOV     -4(R2),R4       ;WORD 2 OF ADB = I
AGAIN:  ADD     R4,R1           ;ADD IT IN
        SOB     R3,AGAIN        ;R1 ← (i-1) + I*(j-1)
SKIP1:  MOV     (R0)+,R3        ;THIRD SUBSCRIPT = k
        DEC     R3              ;  LESS ONE = k-1
        BLE     TEST            ;IGNORE IF ≤ 0
        MOV     -2(R2),R4       ;WORD 3 OF ADB = I*J
AGN1:   ADD     R4,R1
        SOB     R3,AGN1         ;R1 ← (i-1) + I*(j-1) + I*J*(k-1)
;******CONVERT TO BYTE INDEX ******************************************
TEST:   ASL     R1              ;R1 ← 2*R1, TWO BYTES PER WORD
        CLR     R4              ;SET UP ACCUMULATOR
        MOV     -6(R2),R5       ;GET N, SIZE
        BIC     #177400,R5      ;N = LOW BYTE = WORDS/ELEMENT
SCALE:  ADD     R1,R4
        SOB     R5,SCALE        ;R4 ← [(i-1)+I*(j-1)+I*J*(k-1)]*2N
;*******TEST FOR ARRAY BOUNDARY ***************************************
        MOV     -6(R2),R5       ;GET N, SIZE
        SWAB    R5              ;SIZE, N
        BIC     #177400,R5      ;SIZE = N*I*J*K
        ASL     R5              ;2*SIZE (2 BYTES/WORD)
        CMP     R5,R4           ;INDEX TOO GREAT?
        BLOS    YES             ;YES!
OK:     MOV     (R2),(R0)       ;NO, BASE ADDRESS OF ARRAY, M(1,1,1)
        ADD     R4,(R0)         ;ADD IN INDEX TO GET ADDRESS OF M(i,j,k)
YES:    TST     (R0)+           ;BUMP R0 BEFORE RTS R0
        JSR     PC,REST         ;RESTORE R1-R5
        RTS     R0              ;AND RETURN
```

Рис. 5.5. Подпрограмма вычисления адреса элемента массива

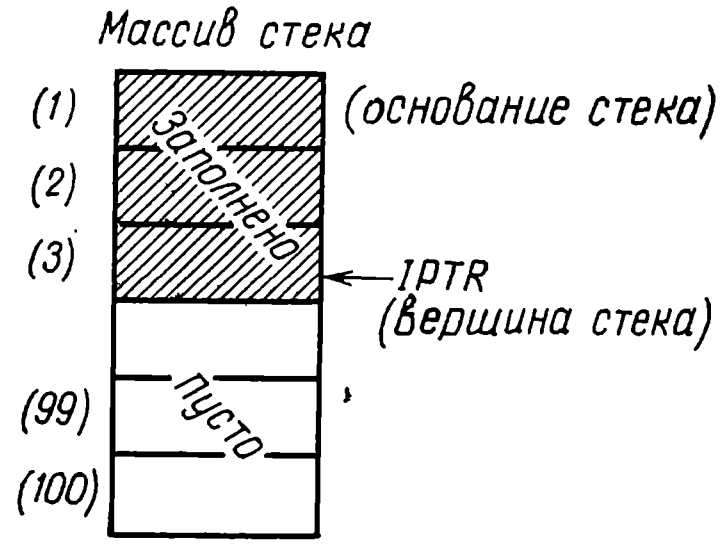
Прежде чем передать управление в вызывающую программу, подпрограмма осуществляет проверку принадлежности вычисленной ячейки к области памяти, отведенной под массив. Если это условие удовлетворяется, она возвращает действительный адрес, в противном случае — код ошибки.

5.6.2. Стеки, двойные стеки и очереди

Читатель уже знаком со *стеком, заполняемым сверху*, или просто стеком, представляющим собой часть аппаратуры PDP-11. В контексте данной главы, посвященной структурам данных, стек рассматривается как специальный вид одномерного массива. Большинство программистов, работающих на Фортране, по-видимому, использовали процедуры занесения и выборки, аналогичные приведенной на рис. 5.6. Физически элементы такой структуры при обращении к ним не перемещаются, меняется только значение указателя на первый доступный из них.

```
SUBROUTINE PUSH(AVALUE)
COMMON STACK(100),IPTR
IPTR=IPTR+1
STACK(IPTR)=AVALUE
RETURN
END
```

```
SUBROUTINE POP(AVALUE)
COMMON STACK(100),IPTR
AVALUE=STACK(IPTR)
IPTR=IPTR-1
RETURN
END
```



Схематическое представление стека на Фортране

Рис. 5.6. Подпрограммы PUSH и POP на фортране

Ранее было показано, как можно применять стек для сохранения и восстановления регистров, в целях обеспечения автоматической вложенности вызовов подпрограмм и написания рекурсивных процедур. Стеки можно встретить и в трансляторах, при компиляции арифметических выражений и генерации кодов программ. Подобные структуры используются и в операционных системах; здесь они служат для хранения списков, информирующих о том, «что случилось последним».

Стек обычно заполняется в одном направлении, однако можно допустить двунаправленное его заполнение, тогда основная стековая структура может быть преобразована в *двойной* стек или двустороннюю очередь. На такой ЭВМ, как PDP-11, где для реализации стеков служат регистры, двойной стек содержит два указателя; один изменяется в сторону увеличения адресов, а другой — в сторону уменьшения.

Двойной стек имеет еще одно качество: если информация из него выбирается, то происходит ее физическое уничтожение. Это означает, что в отличие от операций пересылки или копирования загрузка в двойной стек не порождает копий данных, что больше соответствует интуитивным представлениям о выборке и пересылке информации из некоторого места хранения.

В системе PDP-11 имеется и положительная и отрицательная автоматическая индексация, поэтому реализовать двойной стек

просто. Однако в связи с тем, что в автодекрементном и в автоинкрементном методах адресации изменение состояния регистров выполняется соответственно до и после выборки операнда, возникает необходимость в дополнительных нулевых командах:

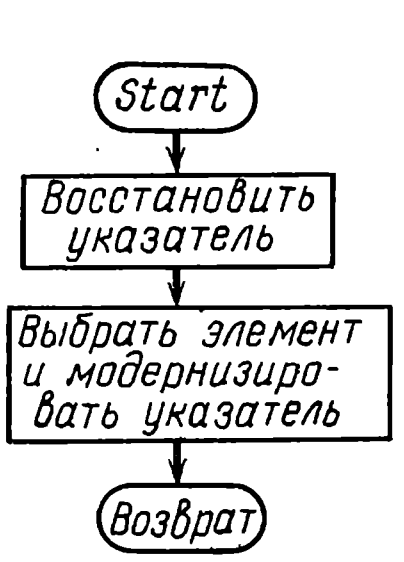
```
TST     -(Rn)
TST     (Rn)+
```

для сохранения правильной позиции указателя стека.

Для работы с двойными стеками имеются четыре основные операции: LSTORE — занесение снизу, USTORE — занесение сверху, LFETCH — выборка снизу и UFETCH — выборка сверху. На рис. 5.7 показаны четыре соответствующие подпрограммы

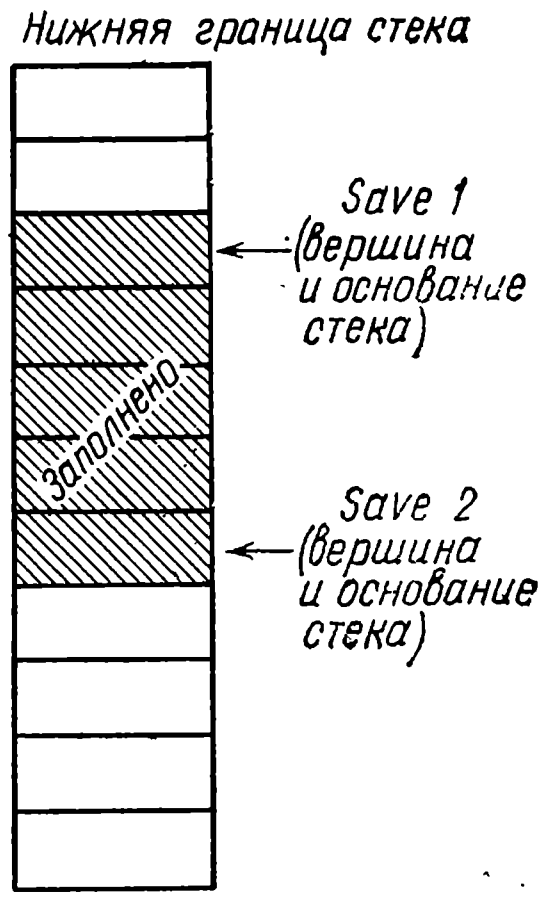
Блок-схема записи

```
LSTORE: MOV     SAVE1,R1
        MOV     R0,-(R1)
        MOV     R1,SAVE1
        RTS     PC
        ;
USTORE: MOV     SAVE2,R1
        TST     (R1)+
        MOV     R0,(R1)
        MOV     R1,SAVE2
        RTS     PC
```



Блок-схема выборки

```
        ;
LFETCH: MOV     SAVE1,R1
        MOV     (R1)+,R0
        MOV     R1,SAVE1
        RTS     PC
        ;
UFETCH: MOV     SAVE2,R1
        MOV     (R1),R0
        TST     -(R1)
        MOV     R1,SAVE2
        RTS     PC
```



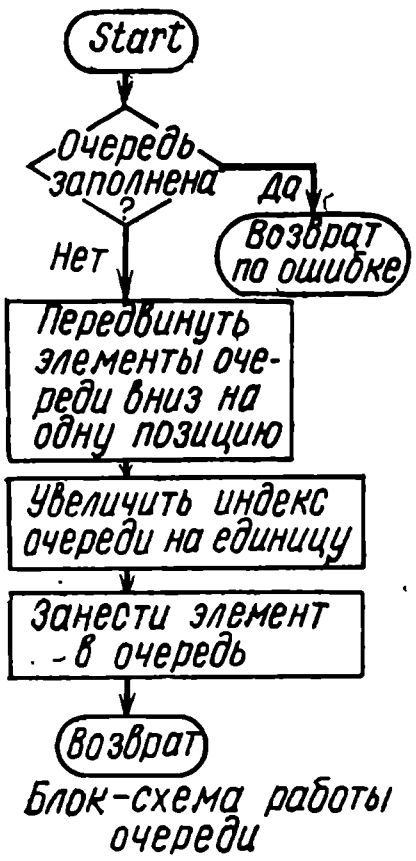
Схематическое представление позиций в стеке

Рис. 5.7. Работа с двойным стеком

Предполагается, что при вызове подпрограмм работы с двойным стеком в R0 записывается либо сохраняемое значение, либо значение, выбираемое из стека. В приведенных программах операции выборки не разрушают выбираемую информацию, в них, как и в обычных стековых операциях, происходит перемещение указателя стека. Разрушение может произойти только при записи новых данных.

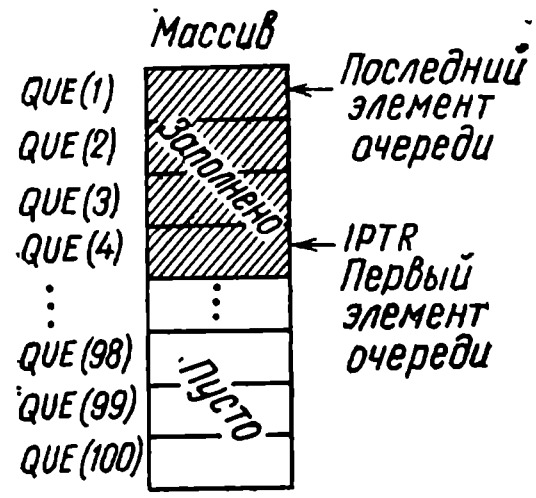
Двойные стеки целесообразно использовать, когда возникает необходимость в заполнении стека с произвольного входа, например если поступающие записи имеют разный приоритет. В системах с разделением времени часто применяется метод циклической

перестановки. В этом случае очередная задача выбирается на обслуживание с одного входа стека, а после истечения отведенного ей времени процессора соответствующий запрос помещается в другой вход. Однако, если необходимо внеочередное обслуживание, запрос помещается не в приемный конец стека, а в выходной.

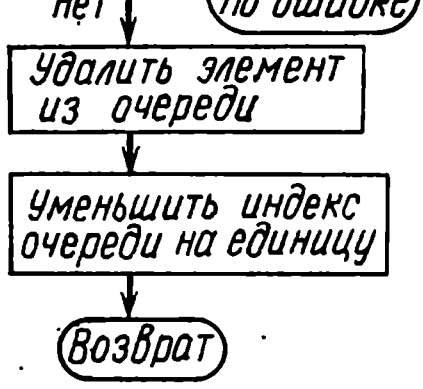


Блок-схема работы очереди

```
      SUBROUTINE QUEUE (X)
      COMMON QUE(100),IPTR.
C     TEST IF QUEUE IS FULL
      IF (IPTR .EQ. 100) GO TO 20
C     NO; PERCOLATE QUEUE
      DO 10 I=1,IPTR
         K=IPTR+1-I
         QUE(K+1)=QUE(K)
10    CONTINUE
C     ENTER ELEMENT AND RETURN
      IPTR=IPTR+1
      QUE(1)=X
      RETURN
20    ERROR CONDITION
      ...
      END
```



Схематическое представление очереди на Фортране



Блок-схема работы двойного стека

```
      SUBROUTINE DEQUEUE (X)
      COMMON QUE(100),IPTR
C     TEST IF QUEUE IS EMPTY
      IF (IPTR .EQ. 0) GO TO 10
C     DEQUEUE ELEMENT AND RETURN
      X=QUE(IPTR)
      IPTR=IPTR-1
      RETURN
10    ERROR CONDITION
      ...
      END
```

Рис. 5.8. Программы оперирования с очередью

Использование двойного стека в режиме, когда записи заносятся через один вход, а выбираются через другой, в значительной мере меняет его характеристики. Положение записи в этом случае становится функцией времени ее запоминания и зависит от состояния соседних элементов. Таким образом, нельзя выбрать запись не в том порядке, который был при занесении. Этот порядок обычно называется дисциплиной *первым пришел — первым ушел* (FIFO), а способ хранения — *очередью*. Как и другие рассматриваемые в этом разделе структуры, очередь — это одна из специальных форм одномерного массива.

Хотя очереди как структуры данных не очень сложны, прибегать к ним надо с определенной осторожностью. Непосредственное

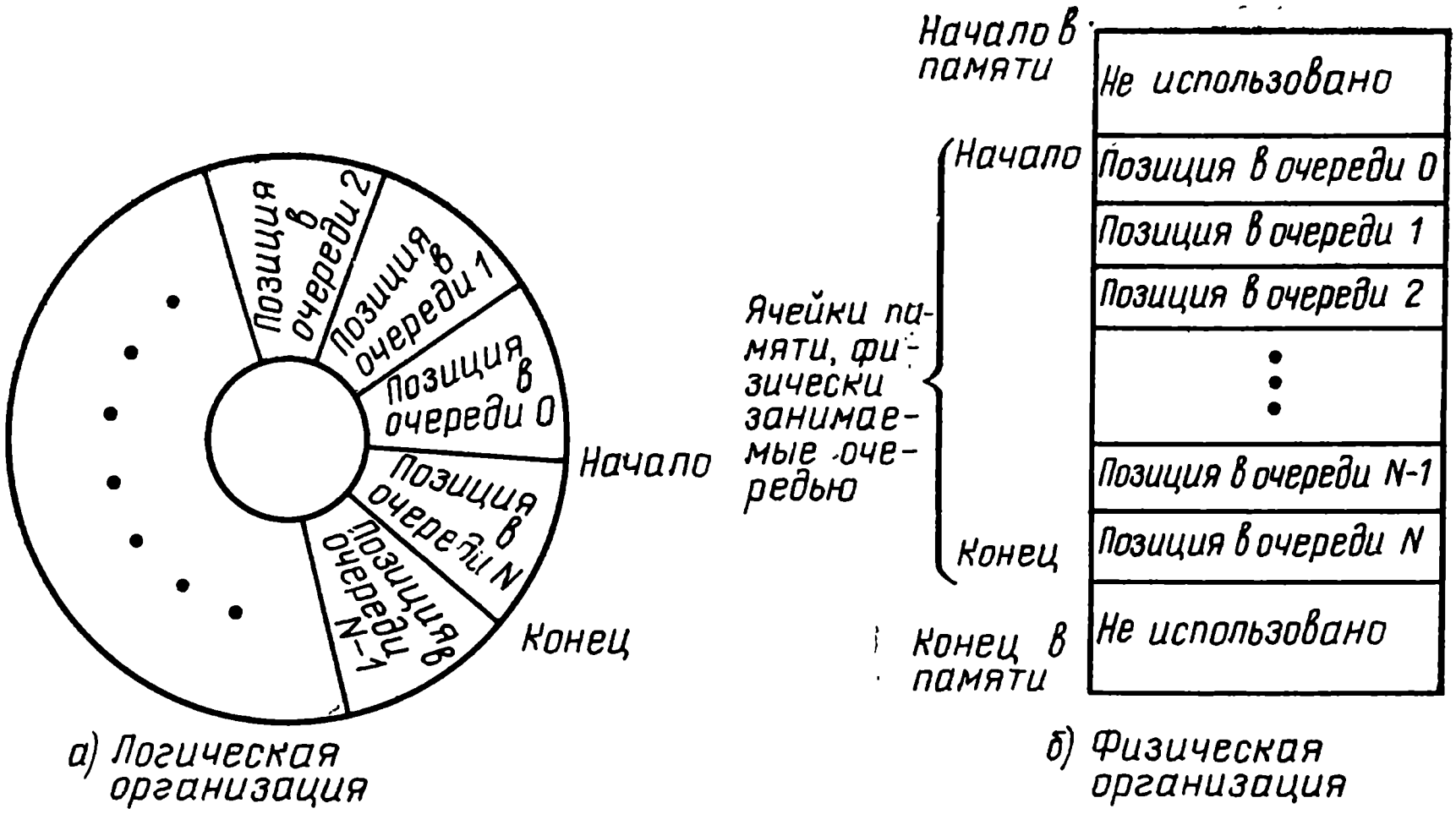


Рис. 5.9. Структура очереди

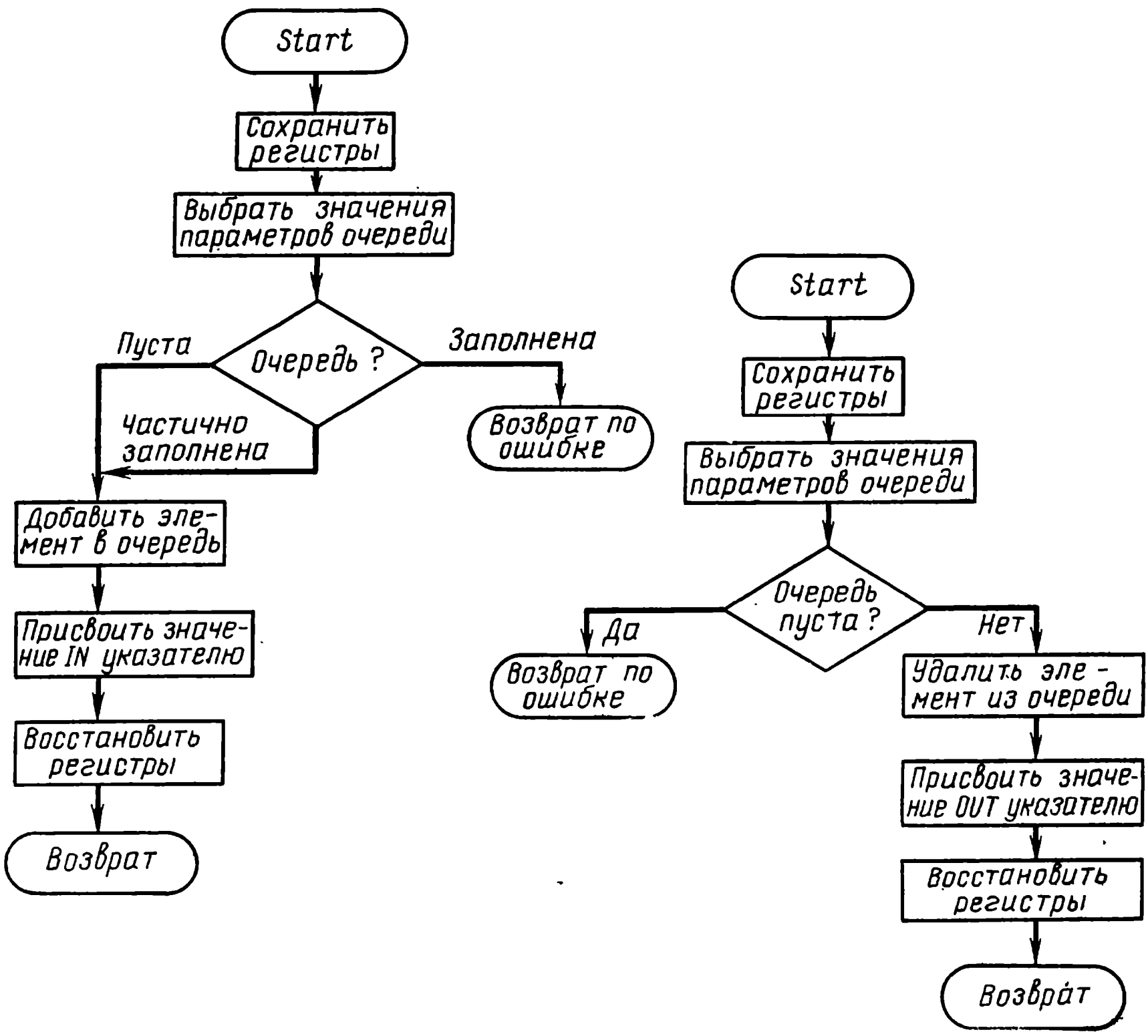


Рис. 5.10. Блок-схемы программ работы с циклическим буфером

использование механизма двойного стека в этом смысле не приводит к желаемому результату. Дело в том, что очередь должна быть фиксированного размера, а записи — последовательно расположены в ней. Основной вопрос состоит в том, должны ли на

```
QUE:    .BLKW 100.              ;QUEUE SIZE=100 WORDS
BEGIN:  0                       ;START OF QUEUE
END:    200                     ;QUEUE LIMIT
IN:     0                       ;IN POINTER
OUT:    0                       ;OUT POINTER
QUEUE:  JSR     PC,SAVE         ;SAVE REGISTERS
;
        MOV     IN,R1           ;FETCH IN
        MOV     OUT,R2          ;FETCH OUT
        MOV     END,R3          ;FETCH END
        CMP     R1,R2           ;TEST IN:OUT
        BEQ     OKAY            ;IN = OUT
        BLO     INV             ;OUT > IN
        SUB     R1,R3           ;END - IN
        ADD     R2,R3           ;END - IN + OUT
        BHI     OKAY            ;TESTS OKAY
INV:    SUB     R2,R3           ;END - OUT
        ADD     R1,R3           ;END - OUT + IN
        BLOS    ERROR1          ;QUEUE FULL
OKAY:   MOV     R0,QUE(R1)      ;PUT ELEMENT IN QUEUE
        TST     (R1)+           ;INCREMENT IN
        CMP     R1,END          ;AT END OF QUEUE?
        BLO     STORE           ;NO
        CLR     R1              ;RESET IN (WRAP-AROUND)
STORE:  MOV     R1,IN           ;SAVE NEW IN POINTER
        JSR     PC,REST         ;RESTORE REGISTERS
        RTS     PC              ;RETURN
ERROR1: ...                     ;QUEUE FULL
        ...                     ;ERROR RETURN
DEQUE:  JSR     PC,SAVE         ;SAVE REGISTERS
        MOV     IN,R1           ;FETCH IN
        MOV     OUT,R2          ;FETCH OUT
        SUB     R2,R1           ;QUEUE EMPTY?
        BEQ     ERROR2          ;YES
        MOV     QUE(R2),R0      ;TAKE ELEMENT OFF QUEUE
        TST     (R2)+           ;INCREMENT OUT
        CMP     R2,END          ;AT END OF QUEUE?
        BLO     STR             ;NO
        CLR     R2              ;RESET OUT
STR:    MOV     R2,OUT          ;SAVE NEW OUT POINTER
        JSR     PC,REST         ;RESTORE REGISTERS
        RTS     PC              ;RETURN
ERROR2: ...                     ;QUEUE EMPTY
        ...                     ;ERROR RETURN
```

Рис. 5.11. Программы работы с циклическим буфером

самом деле элементы очереди перемещаться по ее позициям или для описания процесса перемещения можно использовать какие-либо указатели.

Вначале рассмотрим случай перемещения записей. На Фортране программы для работы с очередями и двойными стеками могут иметь вид, показанный на рис. 5.8. Заметим, что при таком решении не только необходимо перемещать все элементы при добавлении нового, но еще требуется указатель первого элемента очереди в статическом массиве QUE.

Алгоритм циклического буфера представляет другой подход к реализации очереди. При этом структура данных рассматривается в виде цилиндрической поверхности, на которой, как на магнитном диске, элемент с наименьшим адресом следует за элементом с наибольшим. Логическое представление такой очереди дано на рис. 5,9, *а*, а физическое — на рис. 5.9, *б*.

Два указателя — BEGIN (начало) и END (конец) — используются для задания пределов очереди, в то время как два других — IN (вход) и OUT (выход) — указывают на позиции, куда была занесена и откуда была прочитана последняя запись. Если IN = OUT, то очередь пуста, т. е. находится в начальном состоянии. Если к ней добавляется запись, то указатель IN увеличивается, а если выбирается, то — OUT. При IN > OUT область от OUT до IN — 1 содержит элементы очереди. Если же OUT > IN, то в области от OUT до END содержится первая часть очереди, а в области от BEGIN до IN — 1 — оставшаяся часть.

На рис. 5.10 и 5.11 показаны блок-схемы и текст программ для работы с очередями по алгоритму циклического буфера. В них предполагается, что в R0 содержится заносимая в очередь или читаемая из нее запись, а для вызова служат команды JSR PC, QUEUE и JSR PC, DEQUE.

Из двух упомянутых программа QUEUE длиннее, так как для того, чтобы убедиться, что двойной стек пуст, достаточно проверить, выполняется ли отношение IN = OUT. Проверка же заполненности очереди требует в соответствии с рис. 5.12 анализа двух условий — IN > OUT или OUT > IN.

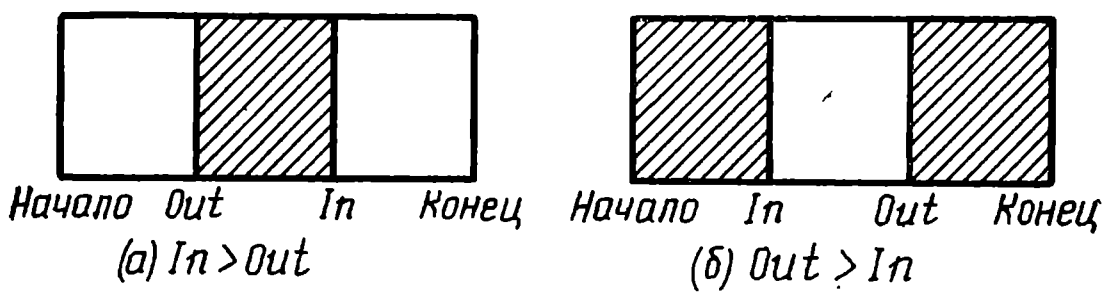


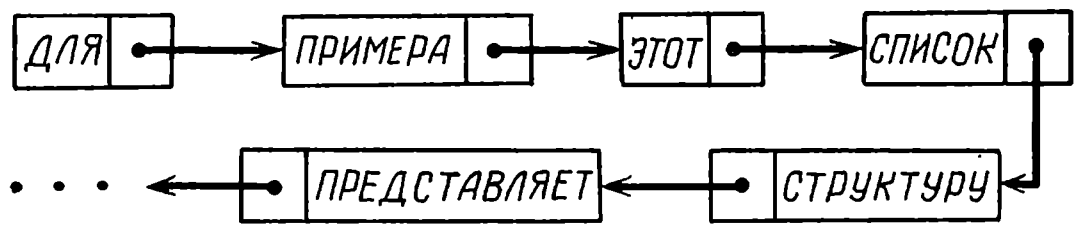
Рис. 5.12. Возможные состояния заполненности очереди

В первом случае (рис. 5.9, *а*) до тех пор, пока END — IN + + OUT > 0, очередь либо частично заполнена, либо пуста. Во втором (рис. 5.9 *б*), при END — OUT + IN > 0 она может быть заполнена только частично. В связи с этим программа QUEUE должна определить, какое условие удовлетворено, и в соответствии с ним проводить анализ.

5.6.3. Списки

Список — еще одна часто встречающаяся структура данных В простейшем случае его можно представить как одномерный массив, для размещения которого в памяти не требуется непрерывной последовательности ячеек. Элементы при этом размещаются в памяти произвольно, но должны быть *связаны* адресными ссылками,

образуя *связанный* список в отличие от непрерывного. Например, списковая структура, представляющая некоторое предложение, может выглядеть так:



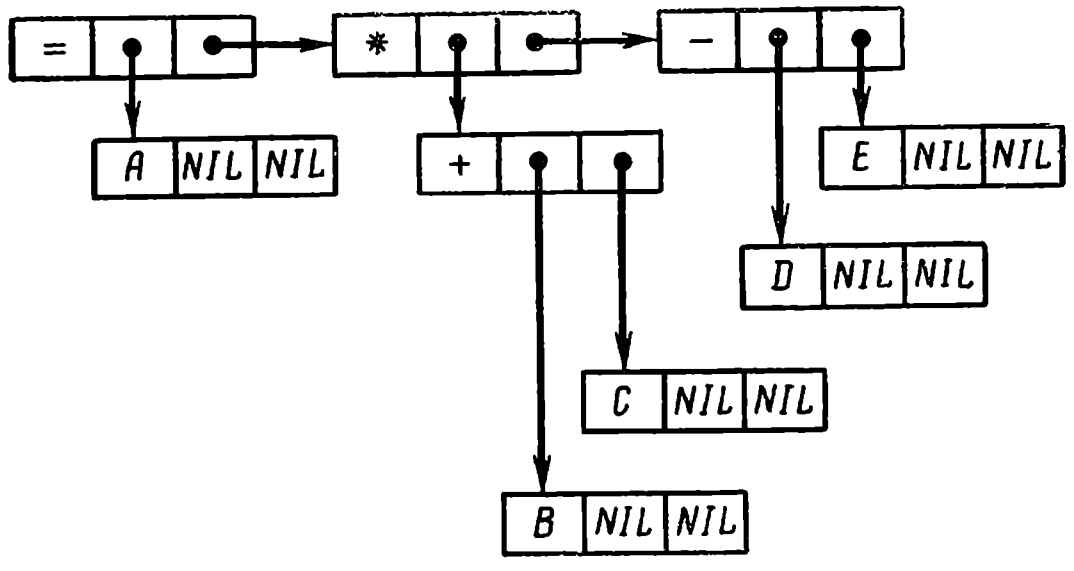
За исключением первого и последнего, любой элемент списка имеет *предшествующий* и *последующий элементы.* Если все элементы списка связаны только с последующими, то такой список называют *односторонним.* В том случае, когда имеется связь как с последующим, так и с предыдущим элементами, список называется *двусторонним.*

Вне зависимости от того, односторонним или двусторонним является список, существует проблема указания его начала, т. е. задания первого элемента. Она решается с помощью указателя, называемого *заголовком,* который ссылается на начало списка. Для удобства используется и второй указатель — *конца списка.* Наличие его не является абсолютной необходимостью, и часто последний элемент списка в поле ссылки на последующий элемент содержит особое значение —NIL, которое обозначает конец списка.

Кроме одномерных существуют и многомерные списки. Интересный пример двумерного списка представляет собой следующее выражение на Фортране:

```
A = (B + C) * (D - E)
```

Оно может быть изображено графически следующим образом:



Каждый двуместный оператор имеет две связи с операндами. В одних случаях операндами могут быть простые переменные, в других — выражения. При разборе такого двумерного списка необходим метод для установления различий между элементами, имеющими одинаковое физическое представление. В одном из методов, пригодном для этой цели, в качестве части списковой струк-

туры используются описатели, которые явным образом характеризуют элементы списка.

Работа с одномерными и многомерными списками осуществляется одними и теми же основными операциями: 1 — создание списка, 2 — добавление элемента, 3 — удаление элемента, 4 — поиск элемента. Поэтому при изучении подобных структур мы используем наиболее простые из них.

При работе на машине PDP-11 проще всего предположить, что элемент списка состоит из двух слов. Первое содержит хранимые данные, а второе — указатель на последующий элемент. Идентификаторами HEAD и TAIL обозначены указатели начала и конца списка. Указатель NIL представляется в виде слова, состоящего из одних нулей. Когда список пуст (начальное состояние), то:

```
HEAD:   0
TAIL:   0
```

Функция программы CREATE состоит в обнулении HEAD и TAIL:

```
CREATE: CLR     HEAD    ;NIL IN HEAD
        CLR     TAIL    ;NIL IN TAIL
        RTS     PC
```

Следующая программа (ADD) должна взять новый элемент и поместить его в список. Она требует два аргумента: 1 — относительная позиция в списке, куда помещается новый элемент; 2 — два добавляемых в список слова. Относительная позиция интерпретируется в соответствии с тем, как это задумано программистом. В данном случае, если значение относительной позиции отрицательное, элемент добавляется в конец списка, а если положительное, то после того элемента, который указан в качестве относительной позиции. Нуль интерпретируется как указание на то, что элемент должен быть помещен в начало списка.

Работа с адресами включаемых в список элементов должна быть заранее предусмотрена программистом, пишущим соответствующую программу. Необходимое распределение памяти, не занятой пользовательской программой или системой, выполняет специальная программа (см. упр. 11 данной главы). При каждом вызове этой программы она возвращает либо адрес выделенной памяти, *либо* флаг, свидетельствующий о том, что свободной памяти нет.

Предполагая, что программист уже использовал программу резервирования и хочет добавить элемент к списку, вызов подпрограммы ADD будет выглядеть так:

```
JSR     R0,ADD          ;CALL ADD SUBROUTINE
(RELATIVE POSITION)     ;FIRST ARGUMENT
(WORD-PAIR ADDRESS)     ;SECOND ARGUMENT
```

Сама программа ADD приведена на рис. 5.13.

На рис. 5.14 показано, как программа ADD осуществляет добавление элемента к списку. Цифрами представлено состояние списка до и после добавления элемента в его середину.

Аналогично выглядят процедуры добавления элемента в начало или конец списка. Соответствующие рисунки, как и программу

```
ADD:    JSR     PC,SAVE         ;SAVE REGISTERS
        MOV     (R0)+,R1        ;RELATIVE POSITION
        MOV     (R0)+,R2        ;WORD-PAIR ADDRESS
        MOV     HEAD;R3         ;GET LIST POINTER
        TST     R1              ;NEGATIVE OR ZERO?
        BMI     END             ;ADD TO END
        BEQ     BEGIN           ;ADD TO BEGINNING
LOOP:   MOV     2(R3),R3        ;GET NEXT ELEMENT
        SOB     R1,LOOP         ;ADD HERE?
        MOV     R3,2(R2)        ;YES, SET UP LINK IN NEW ELEMENT
        MOV     R2,(R3)         ;AND OLD ELEMENT
RET:    JSR     PC,REST         ;RESTORE REGISTERS
        RTS     R0              ;RETURN
BEGIN:  MOV     R3,2(R2)        ;LINK UP NEW ELEMENT
        MOV     R2,HEAD         ;FIX UP HEAD
        BR      RET             ;DONE
END:    MOV     TAIL,R3         ;GET TAIL
        MOV     2(R2),2(R3)     ;LINK IN NEW ELEMENT
        MOV     R2,TAIL         ;FIX UP TAIL
        BR      RET             ;RETURN
```

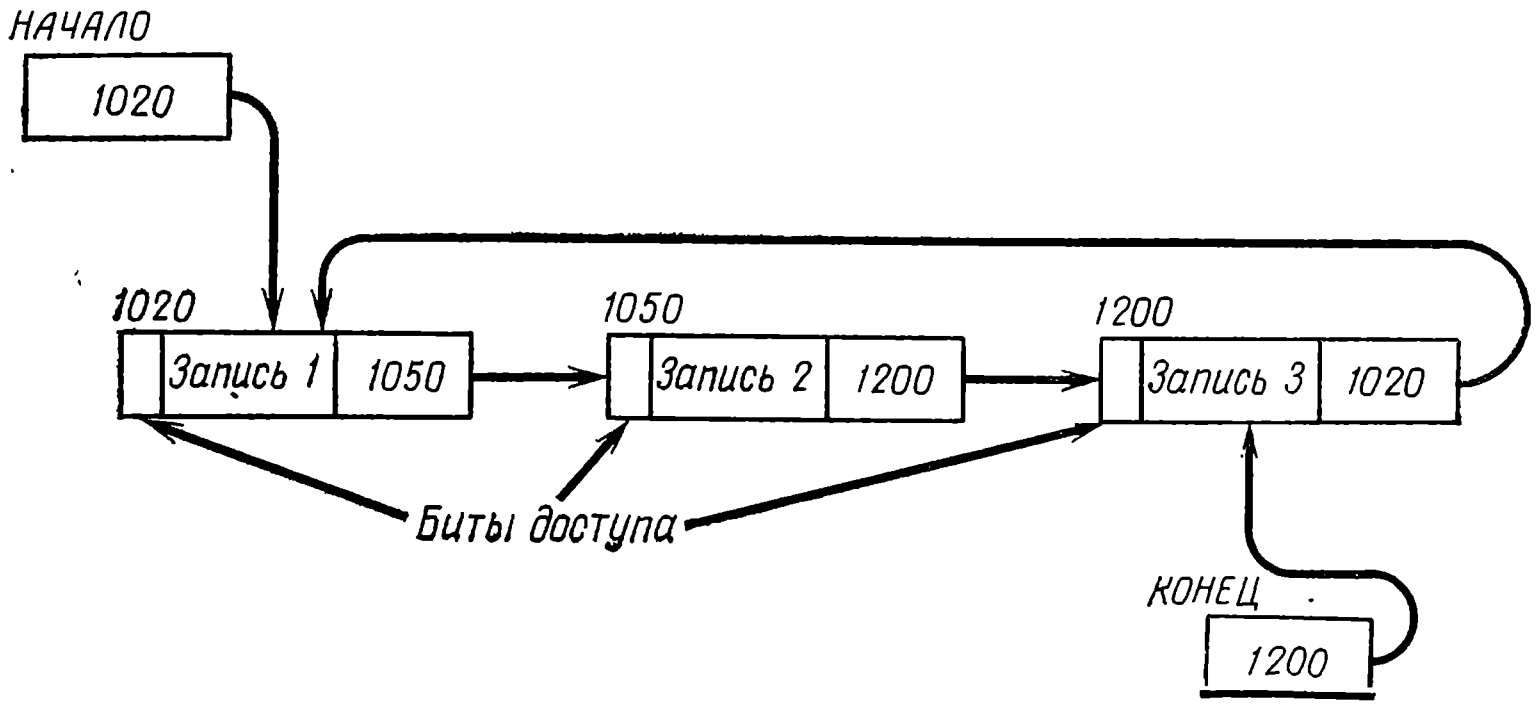
Рис. 5.13. Подпрограмма добавления элемента к списку ADD

удаления элемента DELETE, читатель может самостоятельно выполнить в качестве упражнения. Вызов программы DELETE имеет вид:

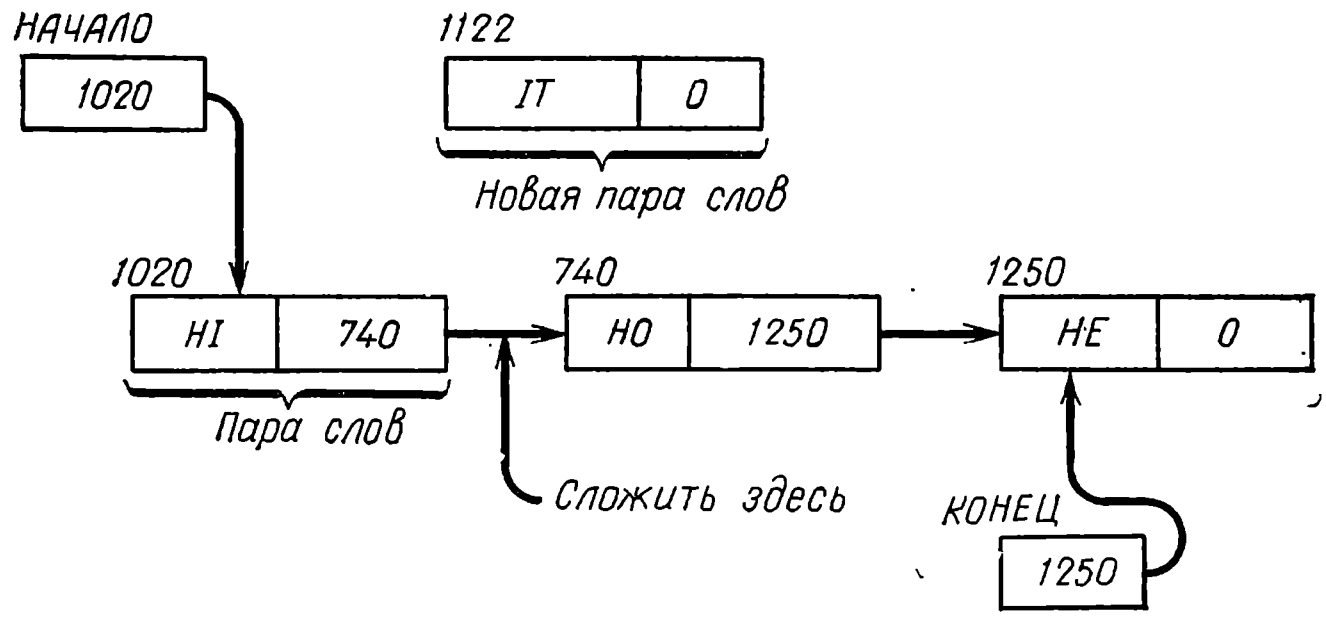
```
JSR     R0,DELETE       ;CALL DELETE SUBROUTINE
(RELATIVE POSITION)     ;FIRST ARGUMENT FOR DELETE
(WORD-PAIR ADDRESS)     ;SECOND ARGUMENT FOR DELETE
```

Процесс удаления элемента требует использования дополнительной процедуры «освобождения» ячеек, ранее занятых элементом, и возвращения их в пул свободной памяти. Если не делать этого, то число занятых, недоступных для использования ячеек будет непрерывно расти. В результате может оказаться зарезервированной вся динамическая память и будет необходимо введение специального средства для *уборки мусора* (очистки списка), для сбора неиспользуемых ячеек.

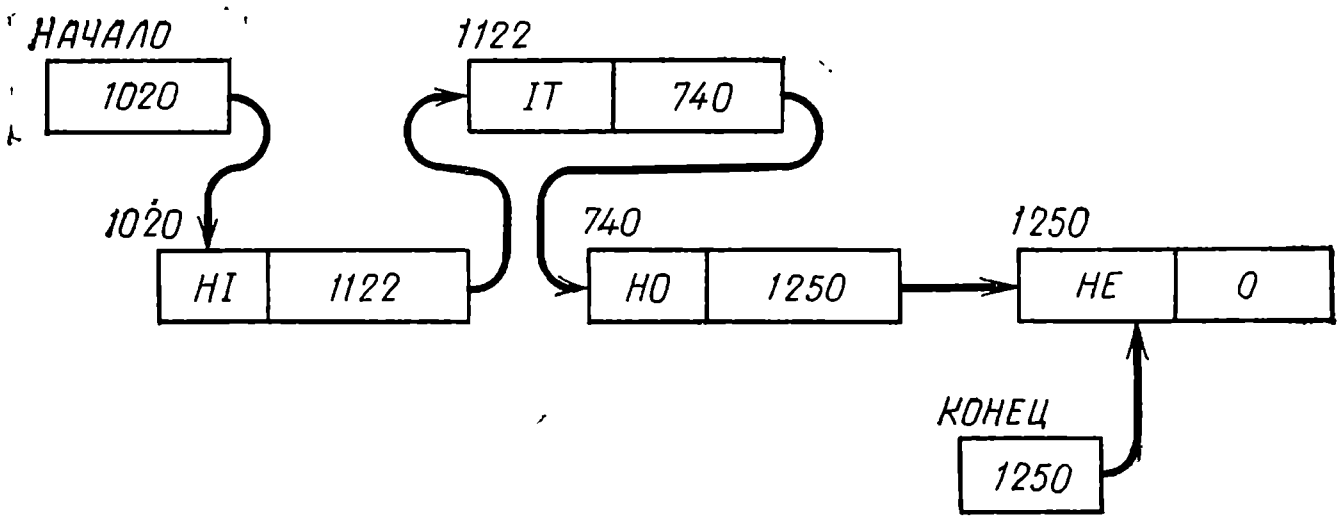
Хотя проблема очистки совсем не тривиальна, для простого списка можно применить следующий прием:



Этот прием требует, чтобы в каждом элементе списка был зарезервирован один бит, называемый *битом доступа.* Всегда, за исключением процесса очистки, этот бит равен 0. При выполнении очистки просматривается весь список и у всех его элементов бит доступа устанавливается в единичное состояние. В результате 0



(а) До



(б) После

Рис. 5.14. Добавление элемента к списку

останется только у тех ячеек памяти, которые на самом деле не принадлежат списку. Обнаружить такие элементы несложно, можно в качестве бита доступа использовать знаковый разряд, рассматривать только неотрицательные элементы и добавлять их в список свободных для повторного употребления. Процедура очистки считается незаконченной до тех пор, пока не восстановлено нулевое состояние бита доступа.

Обращаясь к последней задаче поиска элемента; допустим, что для этого используется программа SEACH. Ее вызов таков:

```
JSR     R0,SEARCH       ;CALL SEARCH
(DATUM)                 ;DATUM TO BE LOCATED
(WORD-PAIR ADDRESS)     ;VALUE TO BE RETURNED
```

На рис. 5.15 представлен текст этой программы.

Из листинга видно, что программа производит поиск по значению и возвращает указатель относительного положения элемента

и адрес двух слов, в которых он содержится. Если элемент не обнаружен, в качестве кода возврата передается 0. Можно создать программу для поиска элемента по номеру в списке. В таком случае параметром вызова будет не его значение, а относительная позиция. Читатель мог заметить, что подобная функция уже выполнялась в программе ADD; для того чтобы ее использовать в целях поиска, нужно внести небольшие изменения.

```
SEARCH: JSR     PC,SAVE         ;SAVE REGISTERS
        MOV     (R0),R1         ;DATUM
        MOV     #1,R4           ;RELATIVE POSITION
        MOV     HEAD,R2         ;START OF LIST
        MOV     TAIL,R3         ;END OF LIST
LOOP:   CMP     R2,R3           ;EMPTY LIST OR NO MATCH
        BEQ     RET             ;NO HIT
        CMP     R1,(R2)         ;TEST FOR MATCH
        BEQ     FOUND           ;GOT A HIT
        MOV     2(R2),R2        ;NEXT ELEMENT
        INC     R4              ;ADD ONE TO RELATIVE POSITION
        BR      LOOP            ;AT END
FOUND:  MOV     R4,(R0)+        ;RETURN REL. POSITION
        MOV     R2,(R0)+        ;AND WORD-PAIR ADDRESS
RET:    JSR     PC,REST         ;RESTORE REGISTERS
        RTS     R0              ;RETURN
```

Рис. 5.15. Программа поиска элемента в списке

Как было показано, существенным достоинством списков является гибкость этих структур. Кроме того, они удобны для представления нечисловой информации при решении задач из области искусственного интеллекта, распознавания образов, при доказательстве теорем, синтаксическом анализе, для построения операционных систем, и т. д. Исходя из этого понимание работы списковых структур существенно для изучения этих областей знаний.

5.6.4. Пример для изучения — реализация ассемблера

Анализ процесса ассемблирования представляет собой хороший пример применения различных структур данных. Входными данными для ассемблера является программа, которую необходимо перевести на машинный язык для выполнения на некоторой вычислительной машине. Традиционно ассемблеры делаются двухпроходными, т. е. программа просматривается транслятором дважды.

На рис. 5.16 в сильно упрощенной форме показаны основные операции *двухпроходного ассемблера.* Из рисунка видно, что ядром процесса ассемблирования является *таблица идентификаторов.* Она представляет собой структуру данных, обеспечивающую преобразование идентификаторов в соответствующие значения. Первоначально в ней содержатся только идентификаторы мнемонических обозначений кодов операций, и в процессе ассемблирования такая таблица расширяется за счет внесения в нее идентификаторов, определенных пользователем.

Таблица идентификаторов. Конструкция таблицы индентификаторов должна быть тщательно продумана в связи с многочисленными обращениями к ней. Например, поскольку идентификаторы кодов операций используются чаще, чем идентификаторы, определенные пользователем, а некоторые коды операций встречаются

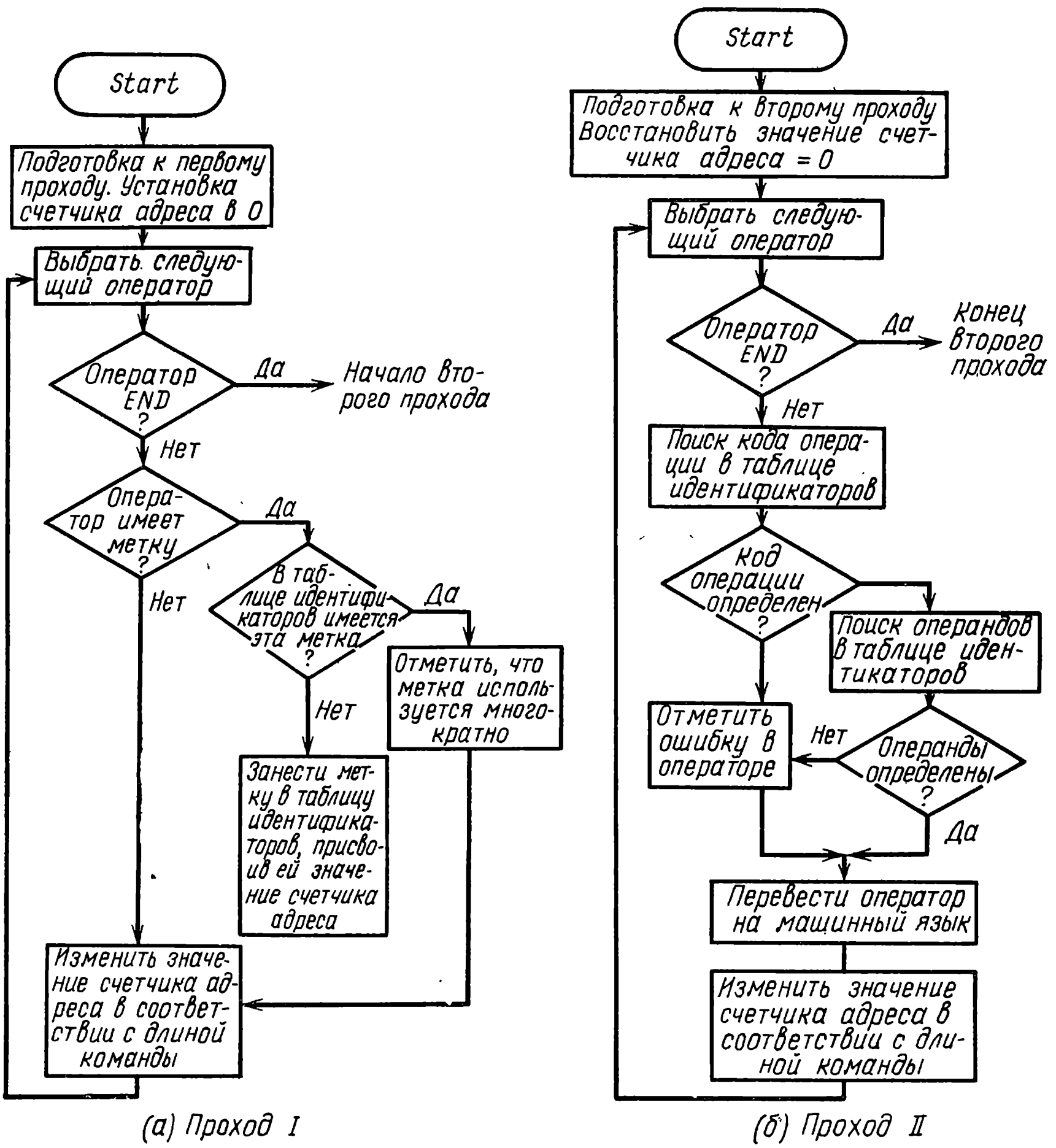


Рис. 5.16. Упрощенная блок-схема двухпроходного ассемблера

чаще, чем другие, упорядочивание идентификаторов в таблице может сократить время поиска в ней.

Если структура таблицы имеет вид простого списка или одномерного массива, то линейный поиск какого-то определенного идентификатора потребует просмотра примерно половины всех элементов. Упорядочение ее может сократить это время, но все же в среднем оно будет достаточно велико. Исходя из этого при конструировании таблицы идентификаторов используются совершенно

иные методы, позволяющие исключить линейный поиск, с которым всегда связаны значительные временные затраты.

Используемый метод называют *хешированием* или *рандомизацией,* он ставит в соответствие идентификатору указатель, адресующий элемент в таблице идентификаторов. Таким образом, хеширование выполняет функцию отображения; она достаточно просто выполняется и обеспечивает однозначное соответствие (во всяком случае, это желательно) идентификатора его позиции в таблице. Например, если применяются 6-байтовые идентификаторы, то простой хеш-функцией может быть сложение по модулю 2 всех 6 байт. Полученное значение используется для обращения в таблицу идентификаторов, аналогично тому, как индекс служит для обращения к массивам.

На Фортране эта функция может быть представлена так:

```
HINDEX = HASH(SYMBOL)
SYMVAL = SYMTBL(HINDEX)
```

где HASH — хеш-функция, SYMTBL — таблица идентификаторов и SYMVAL — значение, соответствующее идентификатору. На языке Ассемблера PDP-11 хеш-функция может быть закодирована следующим образом:

```
HASHER: MOV     #HINDEX,R0      ;R0 ← HINDEX ADDRESS
        CLRB    (R0)            ;HINDEX ← 0
        MOV     #SYMB,R1        ;R1 ← ADDRESS OF Bn, n = 0
HASH:   MOVB    (R1),R2         ;R2 ← Bn
        BICB    (R0),R2         ;R2 ← ~HINDEX ∧ Bn
        BICB    (R1)+,(R0)      ;HINDEX ← ~Bn ∧ HINDEX, n ← n+1
        BISB    R2,(R0)         ;HINDEX ← R2 ∨ HINDEX
        CMP     R1,#SYMB+5      ;ALL SIX BYTES USED?
        BLOS    HASH            ;NO, REITERATE PROCESS
```

На всех моделях PDP-11 имеется встроенная операция сложения по модулю 2, поэтому для общности приведем следующее выражение:

$$A \forall B = (\sim A \wedge B) \vee (\sim B \wedge A)$$

где $\sim A \wedge B$ эквивалентно команде BIC. Как уже было сказано, программа формирует из 6-битовых идентификаторов, состоящих из байт B0, B1, B2, B3, B4, B5, такой результат:

$$HINDEX = (((((0 \forall B0) \forall B1) \forall B2) \forall B3) \forall B4) \forall B5)$$

Это значение хеш-фукнции служит индексом для таблицы идентификаторов, содержащей до 256 элементов (число 256 объясняется тем, что HINDEX — это байт и он может иметь до 256 комбинаций бит).

Две очевидные проблемы связаны с методом хеширования. Во-первых, число генерируемых индексов может быть слишком мало, во-вторых, может оказаться, что одно и тоже значение индекса присвоено разным идентификаторам. К счастью, обе проблемы решаются с использованием одного приема. Если заменить

одномерный массив двумерным, в котором каждому значению хеш-функции соответствует строка, состоящая из четырех элементов, то каждый из них будет соответствовать идентификатору. Визуально эту таблицу можно представить так:

Столбец

Индекс	0	1	2	3
0	элемент 0,0	элемент 0,1	элемент 0,2	элемент 0,3
1	элемент 1,0	элемент 1,1	элемент 1,2	элемент 1,3
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
255	элемент 255,0			

Двумерная схема требует вначале вычисления хеш-индекса, а затем — линейного поиска по строке до тех пор, пока не будет найден нужный или пустой элемент.

Однако такой подход страдает двумя недостатками. Во-первых, в связи с тем, что несколько идентификаторов могут иметь одно и то же значение хеш-индекса, в строке необходимо хранить и значение, присвоенное идентификатору, и сам идентификатор. Во-вторых, для разрешения возможных конфликтов при назначении хеш-индекса требуется зарезервировать достаточное число столбцов, чтобы обеспечить однозначное соответствие между идентификатором и позицией в таблице. Если количество столбцов будет слишком велико, то таблица будет очень слабо заполнена, а если мало, то возникнут ситуации, когда ассемблер не сможет занести в таблицу новый идентификатор. Решение состоит в замене строк списками.

Вместо применения двумерной таблицы можно употребить другой прием; он заключается в том, что к значению хеш-индекса прибавляется константа каждый раз, когда обнаруживается непустой или не совпадающий с данным элемент. Этот прием позволяет после вычисления хеш-индекса выполнять поиск линейно и расширять таблицу. Алгоритм для поиска в такой таблице может быть описан так:

1) вычислить значение хеш-индекса;

2) проверить, совпадает ли элемент с искомым или элемент пуст; если одно из условий выполнено, то прекратить поиск;

3) прибавить смещение к хеш-индексу, проверить, находится ли полученная точка внутри таблицы, если нет, то идти к шагу 5;

4) проверить, что индекс в новой точке совпадает с первоначально вычисленным значением, если нет, то закончить поиск: он безрезультатен;

5) вычесть значение, равное размеру таблицы из хеш-индекса, и перейти к шагу 2.

Для примера предположим, что имеется 16-элементная таблица идентификаторов, смещение равно 5 (смещение должно быть простым числом). Входы в таблицу ранжированы от 0 до 15. Если вычисленное значение хеш-индекса равно 2, то допустима следующая последовательность входов в таблицу:

[2, 7, 12, 1, 6, 11, 0] поиск оканчивается, когда вычисленное значение хеш-индекса = 0

Упаковка элементов в таблице идентификаторов. Одним из элементов в таблице идентификаторов почти всегда должен быть сам идентификатор. Поскольку он может состоять из нескольких символов, желательно упаковать как можно больше их в одном слове. Из прочитанного раньше читатель может сделать вывод, что в таких небольших ЭВМ, как PDP-11, число байт в слове равно 2 и, следовательно, число символов в слове тоже равно 2. Однако это не так.

Число разных символов, с которыми может работать ассемблер, значительно меньше, чем 256 (т. е. 2**8); оно равно примерно 36—40, т. е. 26 символам английского алфавита плюс 10 букв и несколько специальных знаков. Это позволяет упаковать в 16-разрядное слово три символа, используя формулу

$$((C(1) * 50)_8 + C(2)) * 50_8 + C(3)$$

где C — символы в формате *Radix-50*. Ниже приведено соответствие между кодами Radix-50 и ASCII:

Символ	ASCII	Radix-50
Пробел	40	0
A — *Z*	101—132	1—32
$	44	33
.	56	34
Не используется		35
0—9	60—71	36—47

Переведем упаковку идентификатора «SYMBOL», занимающую в коде ASCII 3 слова,

	Нечетный байт	Четный байт	
слово 1	131	123	
слово 2	102	115	= .ASCII /SYMBOL/
слово 3	114	117	

в код Radix-50

S = 073300	B = 006200
Y = 001750	O = 001130
M = 000015	L = 000014
075265	007344
слово 1	слово 2

где тот же идентификатор занимает только 2 слова, что дает значительную экономию памяти.

Существуют два способа для гого, чтобы писать программы упаковки в код Radix-50. Один предполагает использование для каждого символа значения из соответствующей таблицы, второй основывается на приведенной выше формуле. Упрощенная блок-схема алгоритма по второму способу показана на рис. 5.17, а на рис. 5.18 — соответствующая программа. В ней вначале выделяется первый байт из двух слов, содержащих три символа, упаковываемые в код Radix-50. Затем этот байт сдвигается на 3 разряда влево, что соответствует умножению на 8. Этот промежуточный результат запоминается и байт сдвигается еще на 2 разряда; в результате получается C(1)*32. Затем промежуточные результаты суммируются, и получается новый промежуточный результат

$$C(1) * 32 + C(1) * 8 = C(1) * 40 = C(1) * 50_8$$

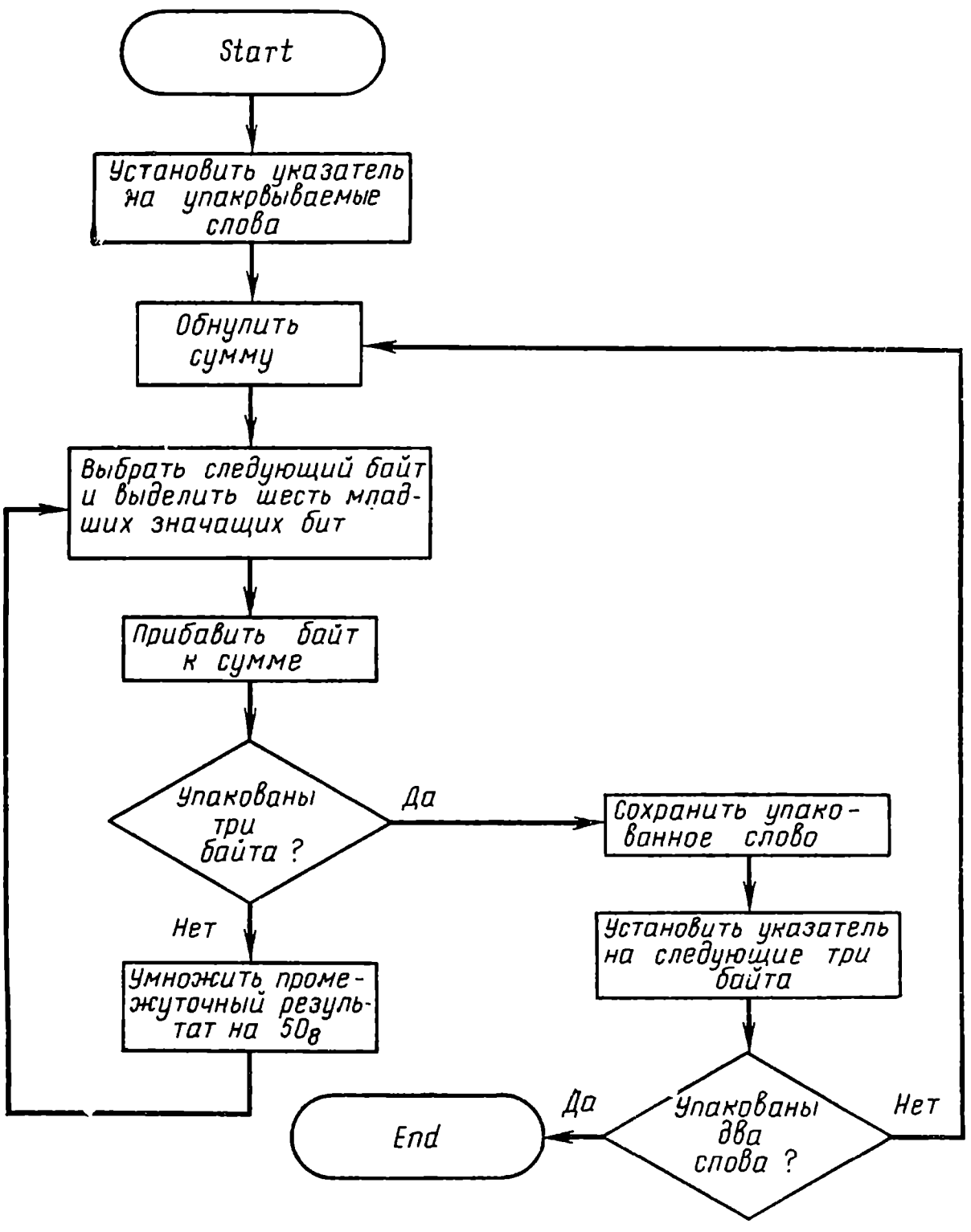


Рис. 5.17. Алгоритм упаковки в RADIX-50

```
PACKER: MOV     #SYMADR,R4      ;ADDRESS OF 1'st WORD TO BE PACKED
        MOV     #SYMADR+2,R5    ;ADDRESS OF 2'nd WORD TO BE PACKED
        MOV     R4,R1           ;INITIALIZE R1
;
PACK:   CLR     R3              ;SUM ← 0
PACK1:
        MOVB    (R1),R2         ;GET CHARACTER
        JSR     PC,CNVRT        ;CONVERT TO RADIX-50
        ADD     R2,R3           ;SUM ← SUM+TRANSLATED CHARACTER
        CLRB    (R1)+           ;CLEAR CHARACTER OUT
        CMP     R1,R5           ;IS R1 ≤ R5?
        BHI     PACK2           ;NOPE, TRIAD COMPLETED
        ASH     #3,R3           ;SHIFT SUM LEFT 3 BITS → C*8
        MOV     R3,-(SP)        ;SAVE PARTIAL RESULT
        ASH     #2,R3           ;SHIFT SUM LEFT 2 BITS → C*32
        ADD     (SP)+,R3        ;SUM ← SUM+PARTIAL RESULT = C*40
        BR      PACK1           ;PROCESS NEXT CHARACTER
PACK2:  MOV     R3,(R4)+        ;STORE RESULTANT TRIAD
        ADD     #3,R5           ;LAST WORD ADDRESS
        CMP     R5,#SYMADR+6    ;DONE?
        BLO     PACK            ;NOPE, REITERATE PACKING
```

Рис. 5.18. Программа упаковки в RADIX-50

При повторении процесса обрабатывается второй байт и получается

$$((C(1) * 50_8) + C(2)) * 50_8$$

Окончательный результат будет получен при прибавлении к нему C(3). Перевод из кода ASCII в Radix-50 и программа распаковки предлагаются читателю в качестве упражнения.

5.6.5. Практическое использование структур данных

Структуры данных играют заметную роль на всех уровнях создания программного обеспечения: при описании, проектировании и, наконец, при реализации всей системы. Необходимость в представлении информации в формате, удобном и для человека, и для машины, реализуется в виде описания типов переменных в языках высокого уровня, очередей, многоуровневых списков и т. д. Поясним это на примере.

Любая мультипрограммная вычислительная система (см. гл. 10) содержит несколько очередей задач, подготовленных к выполнению. Каждый элемент очереди состоит из нескольких последовательных слов, которые содержат: 1 — имя задачи, 2 — ее состояние, 3 — адреса старта и рестарта, 4 — параметры вызова. В случае если задачи группируются в соответствии с их состоянием (например, исполняемые, готовые к исполнению, ожидающие ввода-вывода, приостановленные), применяются многоуровневые списки, подобные тому, который показан на рис. 5.19.

Многоуровневый список сочетает свойства списков со свойствами таблиц, очередей, стеков и двойных стеков. Например, каж-

дая задача описывается при помощи таблицы, которая входит в одну из очередей — выполняемых, готовых, ожидающих I/O или приостановленных. Поскольку задача может быть добавлена в начало или в конец очереди, целесообразно представить эту очередь

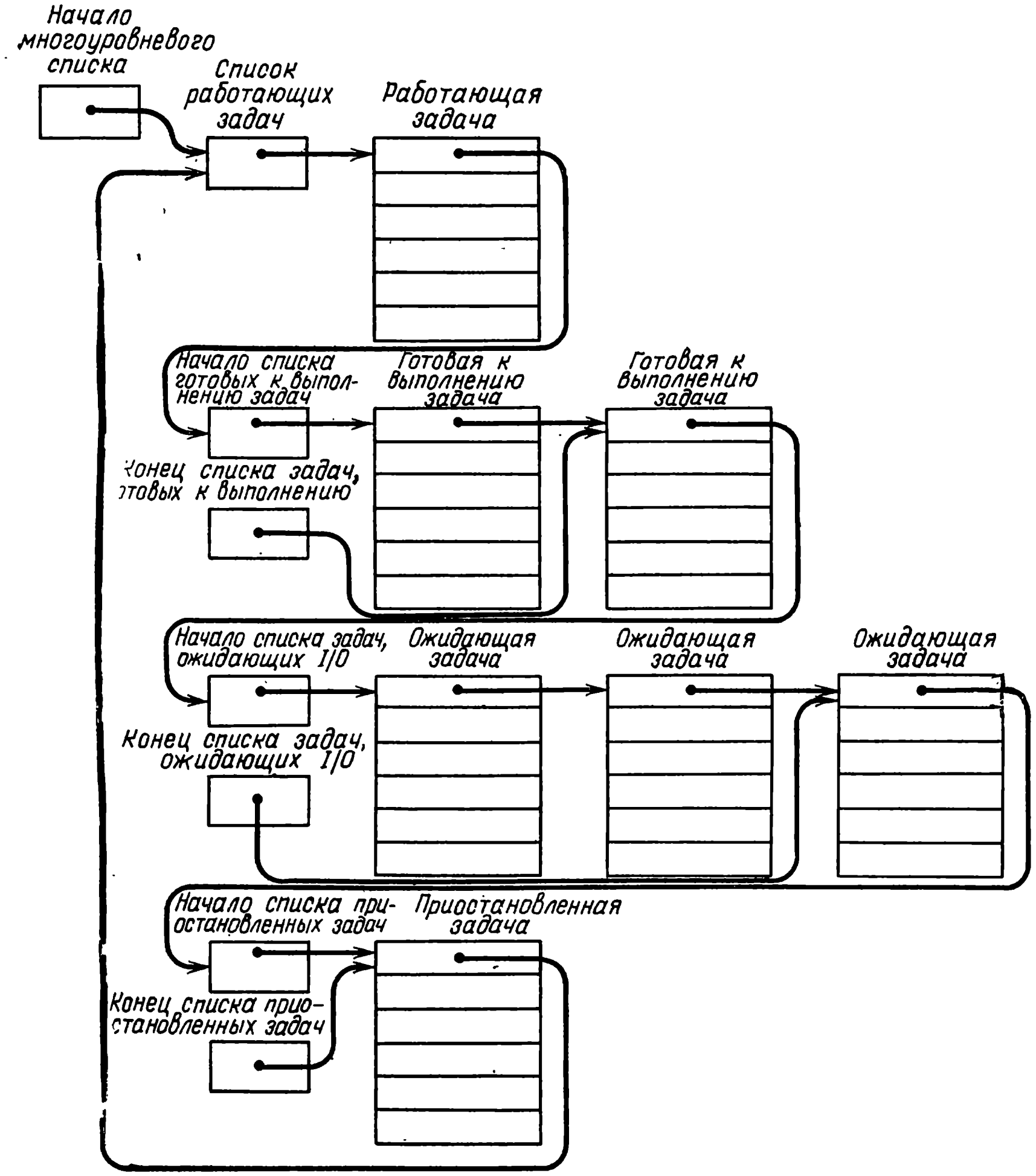


Рис. 5.19. Список задач операционной системы

в виде двойного стека, который допускает использование в качестве обычного, при занесении и выборке записей только с одного его конца. Скажем, если одна программа прерывает другую и информация прерванной должна быть помещена в стек для продолжения ее выполнения в дальнейшем. Однако в целом данные в системе структурированы в форме списка, потому что именно списковая структура наиболее удобна для использования.

УПРАЖНЕНИЯ

1. В чем состоит различие между переносом и переполнением при операции вычитания?

2. Подготовьте таблицу, аналогичную табл. 5.3, используя те же десятичные значения, для выполнения вычитания с двойной точностью.

3. Представьте число 16 768 в форме с фиксированной и плавающей точкой. Длина слова ЭВМ 116 бит, для мантиссы используются 10, а для порядка — 5 бит.

4. Используя 16-битовое слово, определенное в упр. 3, покажите, как может быть представлено отрицательное число с плавающей точкой без использования метода дополнения до 2.

5. При помощи системы команд PDP-11 напишите подпрограмму для сложения по модулю 2 двух значений X и Y, т. е. подпрограмму, реализующую команду XOR, которая имеется в некоторых новых моделях системы PDP-11.

6. Имеется стек, в котором хранится строка символа ASCII, указатель в R0. Напишите подпрограмму для поиска в строке символа «?». Она должна заканчиваться, если обнаружен «?» или нулевой байт.

7. Напишите программу перевода мнемонических обозначений команд работы с кодами условий (CLC, CLV и т. д.) в соответствующие им коды операций. Команды хранятся в произвольном порядке в таблице INST, а эквивалентные им коды операций — в таблице OPS.

8. Как должна быть изменена *программа вычисления адреса элемента массива* для того, чтобы она могла работать со всеми типами массивов, приведенными в таблице параграфа 5.6.1?

9. Напишите на MACRO-11 программы, аналогичные QUEUE и DEQUE, реализованным на Фортране. Сравните их работу с работой программ, использующих круговой буфер.

10. Напишите программу DELETE, описанную в параграфе 5.6.3.

11 Напишите программы ALLOCATE и FREE, описанные в параграфе 5.6.3.

12. Разработайте программу сортировки в алфавитном порядке списка двухсимвольных слов.

13. Перепишите программу ADD, чтобы ее можно было применить для выполнения поиска по относительной позиции.

14. Покажите, как может быть использована списковая структура для реализации очереди.

15. Каким должно быть смещение для 32-элементной таблицы идентификаторов, если для доступа в нее используется метод хеширования. Дайте возможный набор значений индексов для доступа в таблицу, если вычисленное значение хеш-индекса равно 5.

16. Напишите программу для распаковки слов, представленных в формате Radix-50.

БИБЛИОГРАФИЯ

Существует несколько хороших учебников, посвященных структурам данных, в том числе: Berztiss A. T. (1971), Brillinger P. C. и Cohen D. J. (1972), Flores I. (1970), Johnson L. R. (1970), Lewis T. G. и Smith M. Z. (1977) и Maurer H. H. (1977). Конечно, книга Knuth D. E. (1968) остается классическим трудом, одним из немногих, в которых структуры данных обсуждаются в контексте машинного языка и языка Ассемблера. Работа Stone H. S. (1975) также охватывает область структур данных, но примеры в ней приводятся на Алголе. В соответствии с наметившейся тенденцией использования языков высокого уровня для создания и работы со структурами данных ряд вопросов, относящихся к ним, освещен в руководствах по таким языкам, как Снобол, ЛИСП, КОМИТ, Кобол, ПЛ/1, Паскаль и др. Формально модифицированный алгоритм Бута был доказан в статье Rubinfeld L. P. (1975).

Глава 6

МИКРОМАШИННАЯ АРХИТЕКТУРА И МИКРОПРОГРАММИРОВАНИЕ ДЛЯ PDP-11

Хотя мы уже описали применение отдельных элементов аппаратуры ЭВМ (например, арифметическое и логическое устройство) с точки зрения программиста, тем не менее интересно проанализировать их более подробно, чтобы попытаться понять, как реализована PDP-11 на аппаратном уровне. Каждая модель PDP-11 выполнена по-разному, однако самые последние из них имеют много общего, и поэтому изучение микроструктуры можно проводить на одном из образцов. Мы сделаем это на микропрограммной ЭВМ PDP-11/60. Микропрограммирование означает, что *каждая* команда реализуется посредством *многих* микрокоманд, индивидуальных для 11/60. В дополнение к этому пользователь может вводить новые команды, которые становятся индивидуальными для его машины. Машина 11/60 создавалась в расчете на пользователя, владеющего микропрограммированием, и поэтому ее внутренняя архитектура (с точки зрения «микропрограммиста») относительно *проста* для понимания и достаточно *мощная,* чтобы позволить проводить полезные исследования на уровне микропрограммирования. Итак, рассмотрим некоторые характеристики микроструктуры 11/60.

6.1. ВВЕДЕНИЕ В МИКРОПРОГРАММНЫЕ МАШИНЫ

Известно, что при соединении соответствующим образом простых логических цепей можно получить *магистраль данных* — ядро любой ЭВМ. Эта магистраль включает: *комбинационные устройства* (например, сумматоры, дешифраторы), *последовательные устройства* (такие, как счетчики) и *связующие устройства.* Сейчас для удобства мы вновь приведем рис. 2.13 и рассмотрим его под углом зрения простой машинной структуры, изображенной на рис. 2.1. Это позволит более детально изучить компоненты магистрали данных.

В первую очередь вновь остановимся на регистрах (например, IR, MAR, MDR, как это было в разд. 2.5), которые «невидимы» при программировании на уровне машинного языка («макропрограммиста»), но играют такую важную роль при последовательном выполнении команд.

В *микропрограммируемой ЭВМ* последовательность передач от одного регистра к другому (возможно, включающая арифметические или логические операции) определяется *машинными командами* и управляется *микропрограммно.* Как мы увидим, микро-

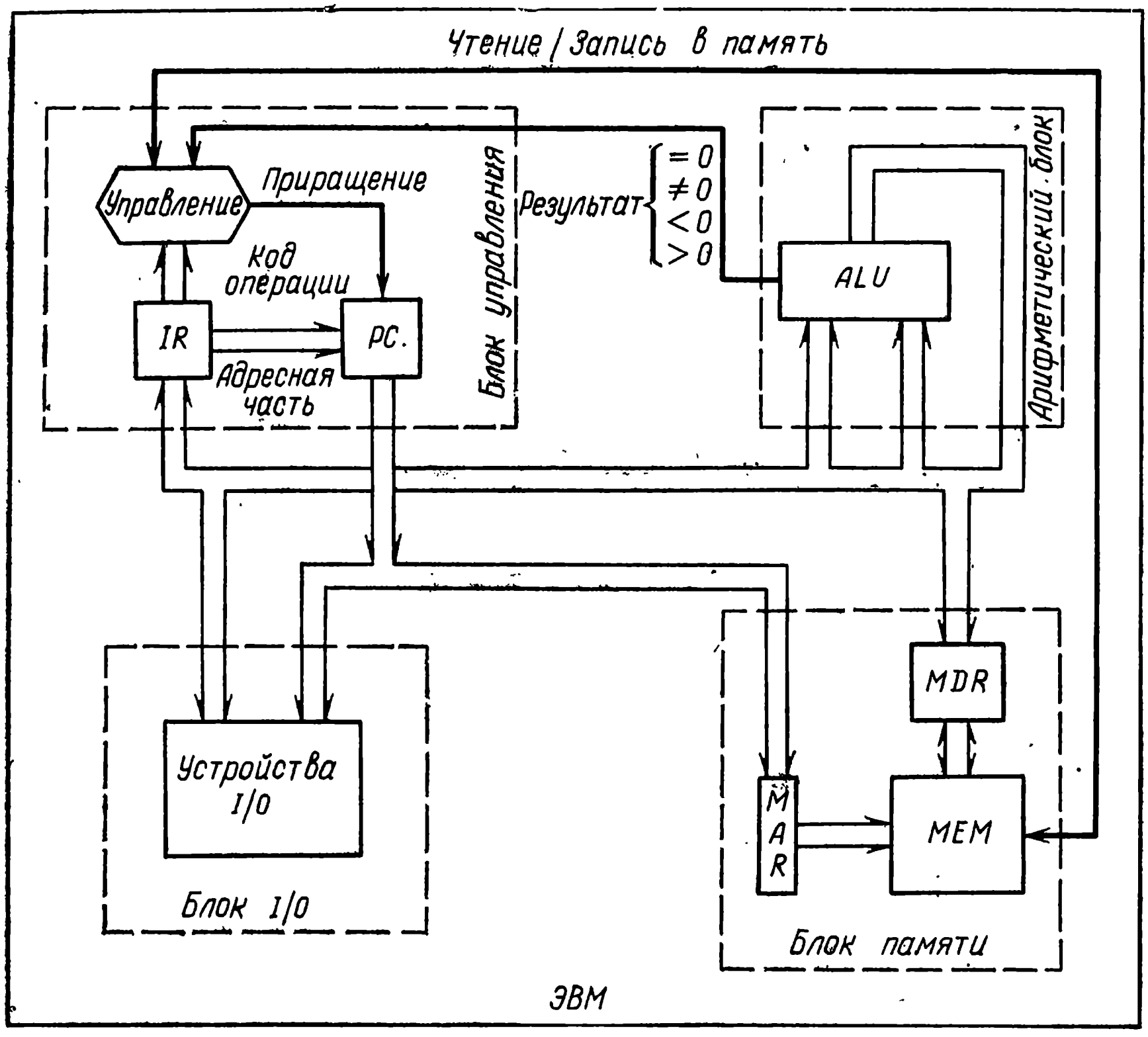


Рис. 6.1. Магистрали данных и управления в простой машине

команды и команды машинного языка близки по форме, но различаются по своим управляющим функциям.

6.1.1. Архитектура и организация

Большинство программ, ориентированных на PDP-11/34, может быть выполнено и на PDP-11/60, так как *архитектура* этих машин идентична (под архитектурой в данном контексте понимается множество ресурсов ЭВМ, которые доступны программисту: система команд, общие регистры, слово состояния процессора и регистры диспетчера памяти). Однако *организация* этих ЭВМ различается тем, что магистраль данных, а следовательно, и микрокоманды, которые управляют ею, индивидуальны для каждой машины.

6.1.2. PDP-11/60 и простая гипотетическая машина

Обычно в подобных работах принято сначала ознакомить читателя с простой гипотетической ЭВМ, описать ее организацию на микроуровне, а затем показать, как выглядят микропрограммы, реализующие простые команды машинного языка. В силу простоты внутренней организации гипотетической ЭВМ этот путь весьма удобен, но не лишен недостатков. Во-первых, аргументов в пользу того, что целесообразно изучать машину, которая никогда не будет реализована и только поверхностно напоминает любую реальную ЭВМ, довольно мало. Во-вторых, принятые упрощения не позволяют демонстрировать интересные и мощные аспекты микропрограммирования.

Мы избрали альтернативный способ. В соответствии с ним ниже будет описано подмножество микромашинной структуры реальной ЭВМ PDP-11/60 и в первую очередь:

1) ресурсы, представляемые машиной, прежде всего те, которые «не видимы» для программиста, применяющего язык машинного уровня;

2) пути и программируемые «переключатели», обеспечивающие необходимую направленность данных между этими ресурсами и между внутренней частью машины и основной памятью;

3) распределение операций и различного рода действий во времени, необходимое для уверенности в том, что желаемая операция выполнена в нужной последовательности.

ЭВМ PDP-11/60 является реальной машиной с помощью которой можно иллюстрировать общие цели и полезность микропрограммирования: выполнение базового множества машинных команд, повышение эффективности выполнения и эмуляция других множеств команд. Магистраль данных машины 11/60 будет описана достаточно подробно, чтобы можно было *наглядно* продемонстрировать все это. Однако еще раз напомним читателю, что *любой* попытке микропрограммирования на 11/60 должно предшествовать изучение соответствующей литературы.

Несколько следующих разделов могут показаться *спорными* (в том смысле, что описание ресурсов дано очень кратко), но они тем не менее весьма *полезны,* так как в них даны обзор внутренней структуры и работы мощной, *реальной* мини-ЭВМ, а также оценка методов реализации и выполнения далеко не тривиальных команд PDP-11. В гл. 9 эти вопросы будут изложены более подробно, чтобы показать, как с помощью микропрограммирования можно улучшить базовое исполнение CPU.

6.2. МАГИСТРАЛИ ДАННЫХ PDP-11/60

Магистраль данных машины 11/60 включает элементы, которые на микроуровне обеспечивают доступ к ресурсам. Управление элементами магистрали данных осуществляется через 48-битовое слово, показанное на рис. 6.2*б*. Для каждого элемента магистрали

X будет описываться его функция, вводные-выводные параметры и управление соответствующим полем микрослова. *В некоторых случаях* мы будем также специфицировать *мнемонику,* которую распознает микроассемблер (точно так же, как обычный ассемблер распознает коды операций и режимы адресации) и в соответствии с ней записывает нужную величину в соответствующее поле микрослова. Таким образом, в следующих нескольких разделах мы изложим «набор исходных средств» для работы микропрограммиста. Они сложнее, а значит, и более мощные, чем те, которыми пользуется макропрограммист. Позже мы покажем, как эти средства применяются для реализации и расширения множества команд PDP-11.

Читая описание каждого элемента, рекомендуем анализировать рис. 6.2а, находить позицию этого элемента и соответствующие пути данных.

6.2.1. Арифметическое и логическое устройство

Ядром магистрали данных является *арифметическое* и *логическое устройство*, или ALU. По каждому из двух портов в ALU может поступать *входная информация* от нескольких регистров:

1. Через порт A из XMUX или CMUX (т. е. от мультиплексоров X или C — 16-битовых регистров), или от одного из шестнадцати 16-битовых регистров быстрой памяти ASP LO или ASP HI. Входной регистр определяется *мультиплексором,* т. е. устройством, подключающим один из n своих входов к одному выходу.

2. Через порт B от одного из шестнадцати 16-битовых регистров быстрой памяти C (CSP), BSP HI или BSP LO.

ALU обеспечивает выполнение базовых арифметических и логических операций, являющихся основой всех вычислений, выполняемых центральным процессором. Как видно из табл. 6.1, 4-битовое управляющее поле ALU (биты 44—47 микрослова) опреде-

Таблица 6.1. Функции и операции ALU PDP-11/60

ALU/Мнемоника	*Определение операции*	*Источник CIN*	*Комментарии*
0 NOT-A	Op ← ~A	1	1's complement
1 A-PLUS-B-PLUS-PS(C)	Op ← A+B+PS(C)	PS(C)	Used for ADC
2 NOT-A-AND-B	Op ← ~A∧B	PS(C)	Used for BIC
3 ZERO	Op ← 0	PS(C)	Generate 0
4 A-PLUS-B-PLUS-D(C)	Op ← A+ B+D(C)	D(C)	Used for FPP
5 A-PLUS-~B-PLUS-D(C)	Op ← A+~B+D(C)	D(C)	Used for FPP
6 A-XOR-B	Op ← A ∀ B	D(C)	Exclusive Or
7 A-AND-~B	Op ← A∧~B	D(C)	Reversed BIC
10 DIVIDE	Op ← A-B; D(C)=1	0	Conditional add/sub,
	Op ← A+B, D(C)=0	0	used in DIV
11 A-PLUS-B	Op ← A + B	0	Add
12 B	Op ← B	0	Select B
13 A-AND-B	Op ← A∧B	0	Logical AND
14 A-PLUS-B-PLUS-1	Op ← A + B + 1	1	
15 A-MINUS-B	Op ← A - B	1	Subtract
16 A-IOR-B	Op ← A ∨ B	1	Inclusive OR
17 A	Op ← A	1	Select A

ляет функцию, которая должна быть выполнена над входами каждой из двух 16-битовых шин, A и B, а также определяет источник переноса (CIN) в ALU. «Op» — это выход ALU; PS(C) — внутримашинный бит переноса, получающийся в ALU, в то время как D(C) — это C-бит, находящийся в PSW.

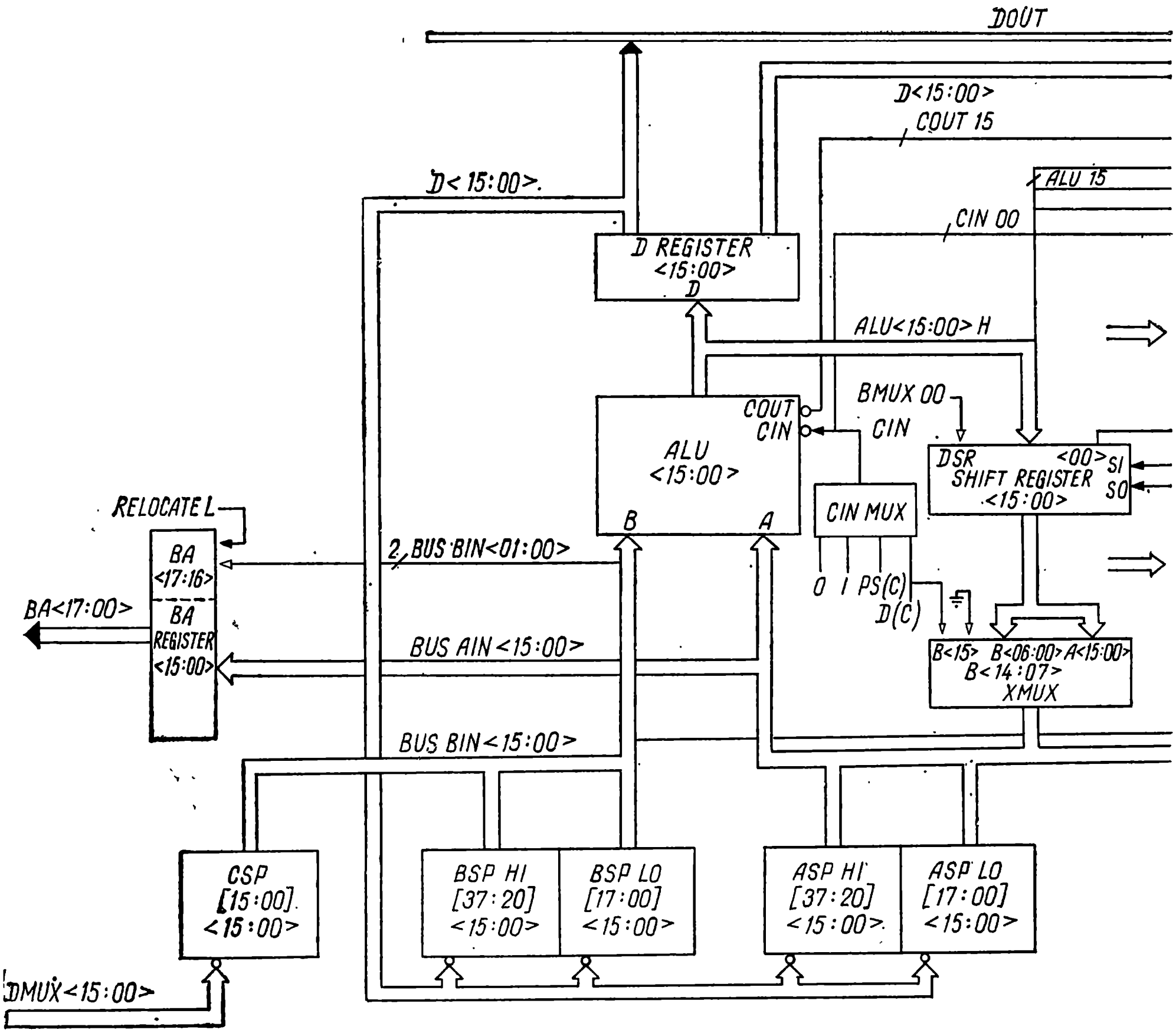
6.2.2. Быстрая память A и B

Каждый блок быстрой памяти (A и B) обеспечивает свою магистраль данных тридцатью двумя 16-битовыми регистрами для хранения данных. В свою очередь такой блок делится на две секции: HI и LO. Так как на шины A и B могут иметь выход и другие регистры, отличные от регистров быстрой памяти, последние должны быть сначала *избраны* или *получить разрешение* прежде, чем их содержимое появится на соответствующих шинах. Выполнение этой функции обеспечивают два 2-битовых микрополя AEN и BEN.

Если установлен *режим прямого адреса,* на что указывают 2-битовые поля выбора режима адреса каждой из магистралей A (ASEL) и B (BSEL), то 5-битовый адрес нужного регистра быстрой памяти определяется с помощью 3-битового поля адреса регистра (RIF) в совокупности со значением микрополей ASEL/BSEL и AEN/BEN. На рис. 6.3 показано использование полей BEN, BSEL и RIF для указания ячейки в быстрой памяти BSP. Подобным же образом происходит выбор ячейки в ASP с помощью AEN и ASEL. Заметим, однако, что наличие только *одного* поля RIF ограничивает число режимов адресации ячеек ASP и BSP.

С другой стороны, ASEL и BSEL могут указывать на то, что для спецификации адреса быстрой памяти вместо микрополя RIF надо использовать 3-битовое поле операнда-источника *или* операнда-приемника текущей команды PDP-11 (т. е. машинной команды), находящееся в регистре команды IR. Регистр IR на рис. 6.2 *не* показан.

Для повышения эффективности работы с адресными режимами PDP-11 необходимо дублировать содержимое общих регистров Поэтому в первые 8 ячеек быстрой памяти A и B помещается содержимое общих регистров. FACi[j], $i = 0, 1, 2, 3$ и $j = 0, 1, \ldots, 5$ есть шесть 64-битовых аккумуляторов (регистров) плавающей точки, используемых инструкциями FPP (процессор плавающей точки). Расположение FAC1[j] и FAC3[j] в быстрой памяти ASP HI и ASP LO соответственно одинаково с расположением FAC0[j] и FAC2[j] в быстрой памяти BSP HI и BSP LO, показанной на рис. 6.3. Если в оборудовании 11/60 имеется ускоритель процессора плавающей точки (FPP), то в регистре WHAMI в ASP устанавливается соответствующий бит и тогда двадцать четыре 16-битовых регистра становятся доступными для пользователя. Биты в WHAMI обеспечивают выбор оборудования в 11/60 (например, кэш-память включить/отключить, ускоритель FPP



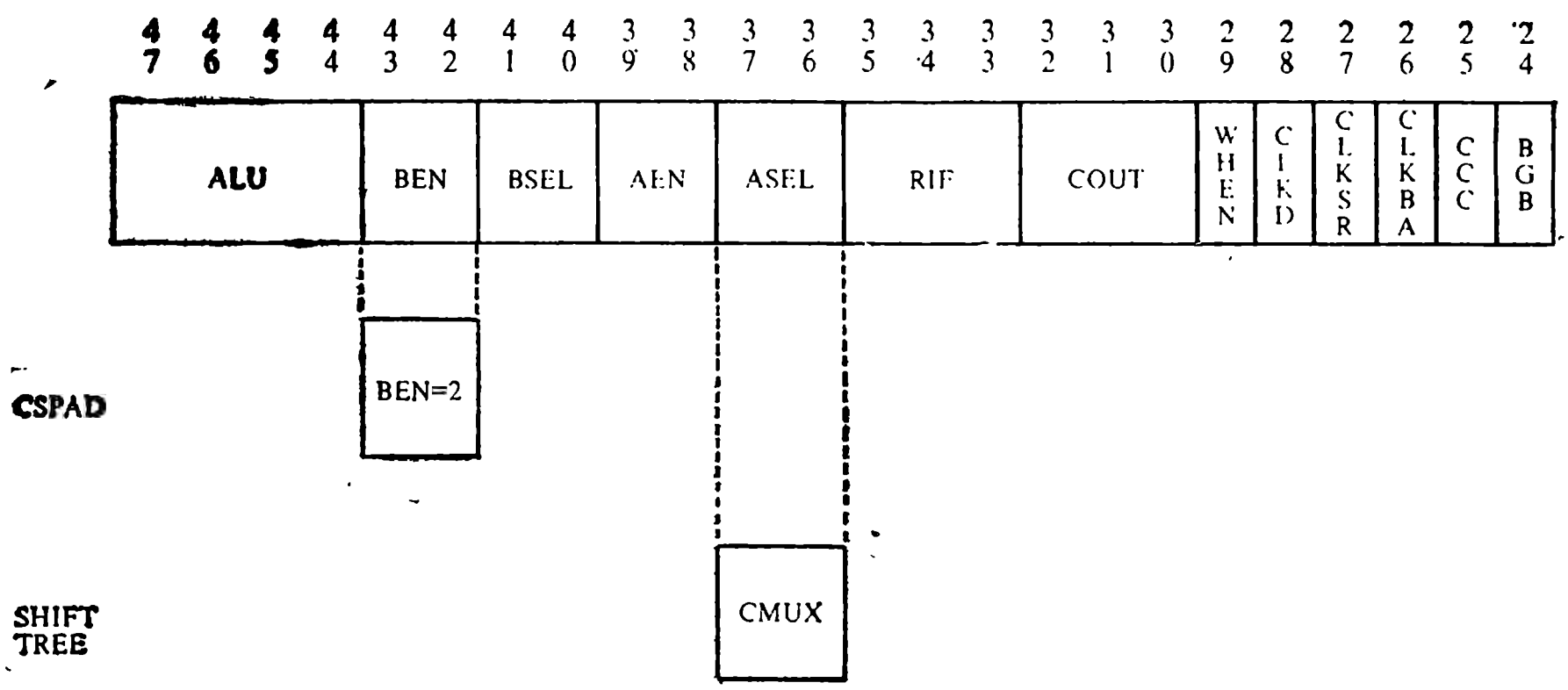


Рис. 6.2б. Упрощенная версия микрослова PDP-11/60,

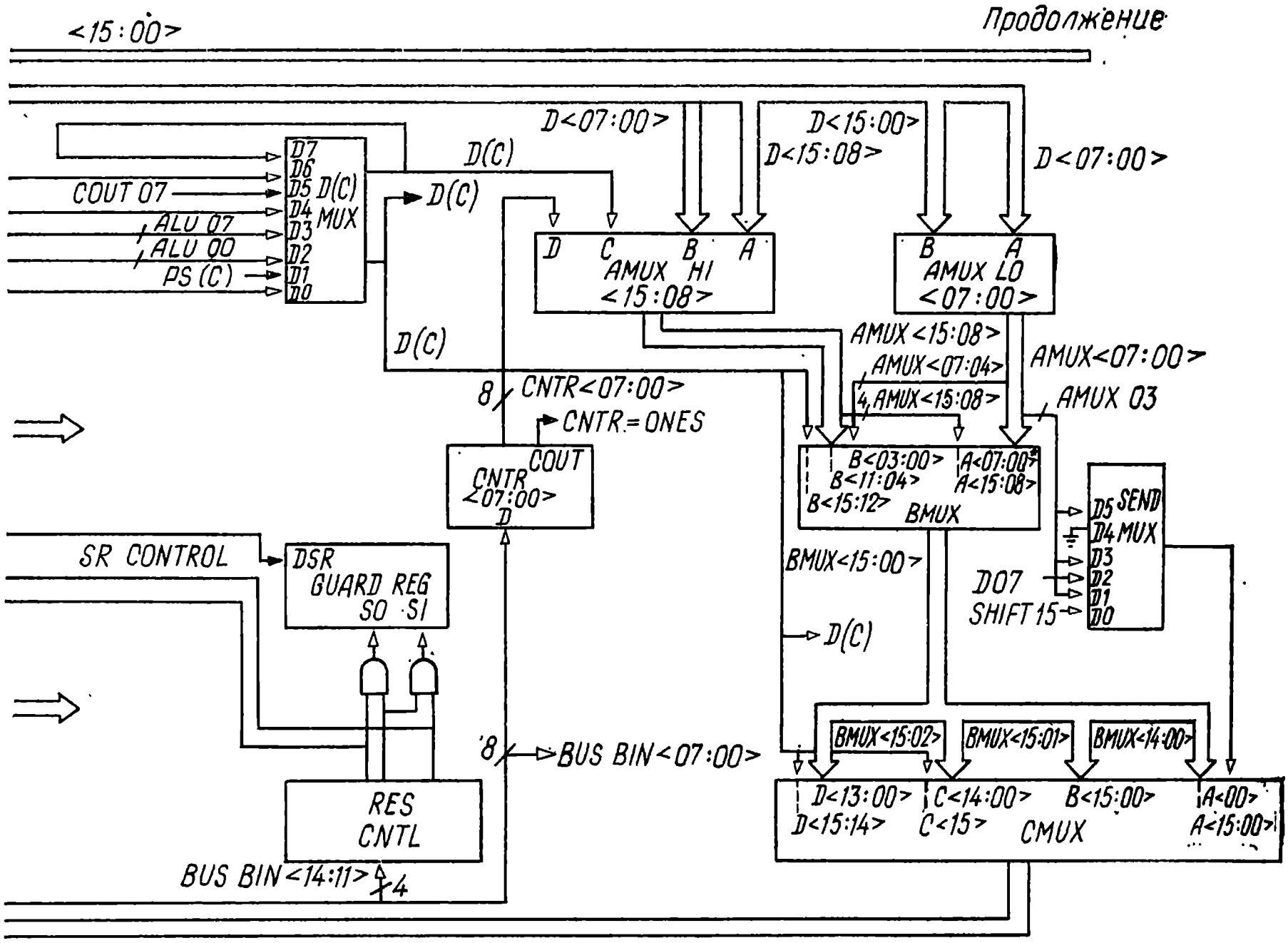
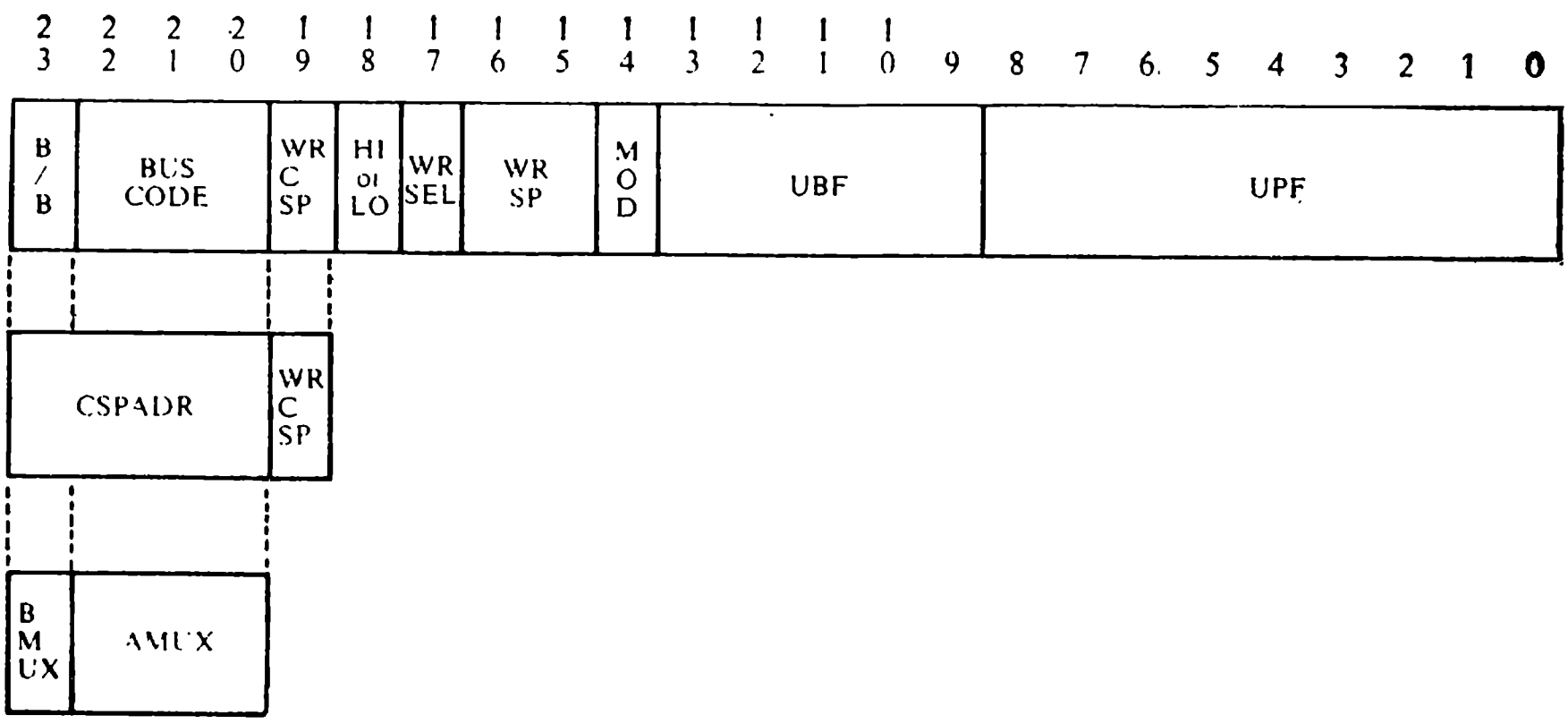


Рис. 6.2а. Упрощенное представление магистралей данных PDP-11/60: ⟨$m:n$⟩ означает позиции бит от m до n включительно внутри слова; [$m:n$] означает позицию слова в быстрой памяти



показывающая поля, обсуждавшиеся в этой главе

подключить/отключить и т. д.). Установка этих бит позволяет следить за состоянием оборудования, и если произойдет сбой, то можно отключить кэш-память и ускоритель. При этом уменьшится скорость выполнения, но функционирование системы не нарушится.

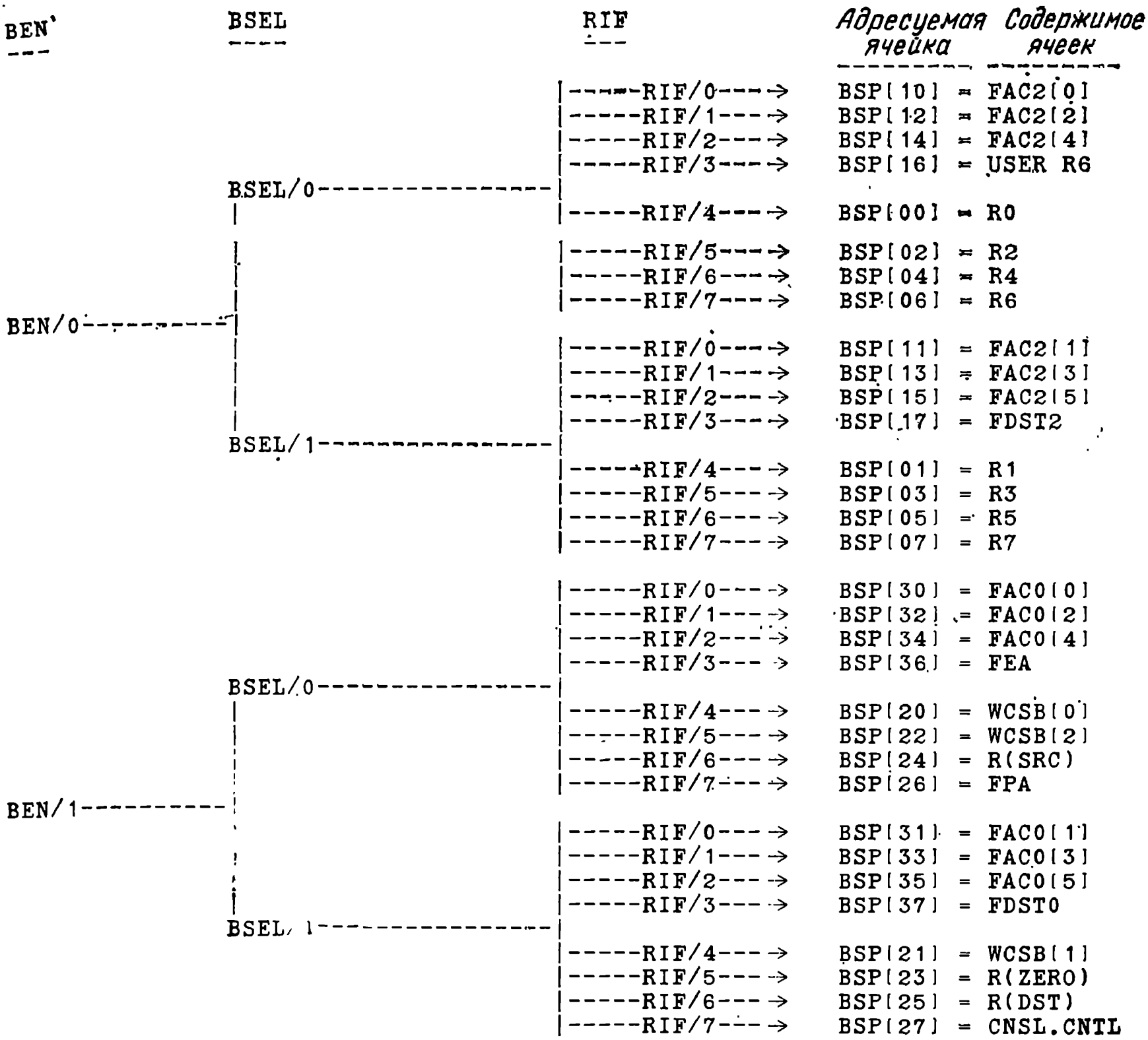


Рис. 6.3. Применение микрополей BEN BSEL и RIF для адресации в BSP

6.2.3. Быстрая память C

Третий потенциальный источник данных для шины B — это 16-словная быстрая память C(CSP), избираемая в случае, если BEN = 2. CSP — окно в главную память (через CSP [15], регистр данных памяти или MDR), а также память для регистрации ошибок. В CSP 16, 17, 14 постоянно хранятся константы 0, 1 и 2 соответственно. Они используются для обнуления ячеек, для манипуляции содержимым регистров в автоинкрементном/декрементном режимах. Если BEN =3, а BSEL = 0, 1, 2 или 3, то эти значения полей также позволяют обратиться к CSP[17], CSP[16], CSP[15], CSP[14] соответственно.

6.2.4. Элементы ввода-вывода ALU

1. **D-регистр.** Вывод ALU помещается на хранение в 16-битовый регистр, если и только если это специфицировано однобитовым микрополем CLKD в момент времени, определенный другим однобитовым микрополем, а именно полем WHEN (разд. 6.3). D-регистр хранит вывод ALU для применения в будущем.

2. **CIN, D(C).** Как было замечено раньше, для учета переноса в ALU имеется один вход — CIN. Через него поступают константы 0 и 1, бит переноса из PSW — D(C) или перенос, полученный в ALU — PS(C). Если же D тактируется, то D(C) может устанавливаться от следующих источников: (1) от бит 0, 7 или 15 вывода из ALU, (2) от выхода селектора CIN, (3) в результате выполнения текущей операции над словом или байтом в ALU, (4) от PS(C) и (5) от предыдущего D(C). Действительный источник установки D(C) определяется значением 3-битового микрополя COUT.

3. **Регистр сдвига и X мультиплексор.** Регистр сдвига (SR) — это 16-битовый регистр, который может загружаться из ALU. X мультиплексор (XMUX) — это коммутатор, который направляет содержимое регистра SR или специального слова (слова, в котором биты 0—6 равны битам 0—6 регистра SR, биты 7—14 равны 0 и бит 15 равен D(C)) на шину A. Отдельного управляющего поля XMUX нет, оно совмещено с полем ASEL. Таким образом, когда AEN равно 2 или 3, ASEL функционирует согласно описанному в параграфе 6.2.2. Когда AEN = 0, младший бит ASEL становится XMUX. SR может совершать сдвиг 16-битовых чисел вправо или влево на один бит. Совместно с D-регистром SR может использоваться для осуществления сдвигов 32-битовых чисел (параграф 6.4.2).

4. **Дерево сдвига.** Дерево сдвига (ST) представляет собой комплекс, включающий три мультиплексора: AMUX, BMUX и CMUX. ST работает в зависимости от выхода D-регистра и указывает, какие операции надо выполнить над 16-битовыми числами: сдвиг на один бит влево, сдвиг на n бит вправо с учетом или без учета знака ($0 < n < 15$), перестановка байта. AMUX, BMUX и CMUX управляются микрополями шириной 3(AMUX), 1(BMUX) и 2(CMUX) бита соответственно. Заметим, что микрополя AMUX и BMUX пересекаются с битами селектора CSP, CSPADRS. Таким образом, если ST задействовано, то можно получить доступ только к четырем ячейкам базовых констант CSP при условии, что BEN = 3.

В параграфах 6.4.1 и 6.4.2 более подробно будет сказано о структуре и применении дерева сдвига и регистра сдвига.

6.3. МИКРОЦИКЛЫ

Электрические сигналы распространяются по проводам с некоторой скоростью. В свою очередь, активные и пассивные комплекты электронных устройств также требуют конечного времени

для преобразования входных сигналов в соответствующие выходные. Поэтому каждый микроцикл 11/60, равный 170 наносекундам (1нс = 10 ** — 9 с), делится на четыре временных такта, отмечаемых четырьмя «импульсами»: P1, P2, P3 и мP3 (микроP3). Формально микроцикл — это интервал между двумя следующими друг за другом задними фронтами импульсов мP3.

6.3.1. Совмещение операций

В течение одного микроцикла может выполниться более чем одна операция. При конструировании микропрограмм определяются моменты, когда должна осуществиться та или иная из них. Например:

1. Результат работы ALU может тактироваться или загружаться в D в момент импульсов P2 или P3, в зависимости от того, чему равно WHEN. Если WHEN = 0 (мнемоника P2-T), то происходит тактирование, если же WHEN = 1 (P3-T), то происходит загрузка в D. Во всех этих случаях поле CLKD = 1. Как было замечено ранее, D(C), избираемое на основании значения поля COUT, тактируется в то же самое время, что и D. Точно так же, если CLKSR = 1, то SR тактируется одновременно с D, и будет производить сдвиг на единицу вправо, на единицу влево или загружаться содержимым выхода ALU, в зависимости от состояния 2-битового поля в регистре RES (разд. 6.6).

2. Во время этого же микроцикла выход D может быть записан в одну из ячеек ASP, BSP и/или CSP, если и только если D тактируется в момент P2. Выбор ячейки быстрой памяти производится по содержимому полей управляющего микрослова. Покажем это на конкретном примере, в котором мнемоника выполняемых функций дается в круглых скобках.

```
MOD    = 0, enable write to ASP and/or BSP (CLKSP)
       = 1, disable write to ASP and/or BSP (NOCLK)

WRSP   = 0, no write (NOP)
       = 1, write to ASP, if enabled as per MOD (WR-A)
       = 2, write to BSP, if enabled as per MOD (WR-B)
       = 3, write to ASP & BSP, if enabled as per MOD (WR-A-AND-B)

WRSEL  = 0, use ASP read address as write address (A-ADDR)
       = 1, use BSP read address as write address (B-ADDR)

HI/LO  = 0, write LO section of selected scratchpads (LO)
       = 1, write HI section of selected scratchpads (HI)

WRCSP  = 0, do not write to CSP (NO)
       = 1, write to CSP (YES)
```

Здесь отражены многочисленные события, происходящие во время микроцикла, а следовательно, и возможности, доступные «микропрограммисту». Строки кодов, которые присваивают управляющему полю одной и той же инструкции некоторое значение, заканчиваются запятой:

Машинная команда:

```
ADD Rn,Rm    (assumed resident in IR)
```

Микрокоманда:

```
ALU/A-PLUS-B,
BEN/BSPLO, BSEL/SF,
AEN/ASPLO, ASEL/DF,
CLKD/YES, WHEN/P2-T,
MOD/CLKSP,
HILO/LO,
WRSP/WR-A-AND-B,
WRSEL/A-ADDR
```

Это означает настроить ALU на выполнение операции ADD, использовать поле SRC(SF) машинной команды для выбора BSP-операнда и ее поле DST(DF) — для выбора ASP-операнда. Оба операнда находятся в *младших* адресах быстрой памяти. Сформировать результат в регистре D в момент P2. Подготовить *младшие* адреса быстрой памяти для записи. Выполнение микрокоманд требует, чтобы команда «ADD Rn, Rm» присутствовала в регистре IR. Следовательно, она должна быть предварительно загружена в этот регистр. Кроме того, необходимо также установить соответствующие биты тех управляющих полей микрослова, мнемоника которых заканчивается наклонной чертой «/». Простой заменой A-ADDR на B-ADDR в WRSEL мы получим некоторое изменение в выполнении команды ADD Rn, Rm, заключающееся в том, что результат операции сложения будет помещен в поле операнда-источника.

3. Так как *выходные данные*, полученные в результате срабатывания дерева сдвига (т. е. CMUX), могут применяться в качестве *ввода* по шине A, то над данными, размещаемыми в регистре D в конце одного микроцикла, в *следующем* микроцикле могут выполняться: (а) сначала операции, обусловленные деревом сдвига с передачей результата в ALU по шине A; (б) затем в ALU может быть произведена операция над данными, поступающими из шины B и результатом, полученным в пункте (а); (в) результат, полученный в ALU, формируется в D в момент P2; и, наконец, (г) содержимое регистра D записывается в одну или обе области быстрой памяти в момент P3.

6.4. ДЕРЕВО СДВИГА И РЕГИСТР СДВИГА: СТРУКТУРА И ИСПОЛЬЗОВАНИЕ

6.4.1. Дерево сдвига

Как было замечено ранее, дерево сдвига состоит из трех мультиплексоров: AMUX, BMUX и CMUX:

1. AMUX. AMUX делится на два байта: AMUX LO и AMUX HI, каждому из которых присваивается значение младшего и старшего байтов регистра D [D(LO) и D(HI)] соответственно. На вход мультиплексора AMUX поступает также выход 8-битового регистра-счетчика (CNTR) и D(C), копируемый 8 раз. Таким

образом, благодаря одному биту поля AMUX, управляющему AMUX LO, и двум битам поля AMUX, управляющим AMUX HI, существуют следующие 8 возможностей выхода поля AMUX:

Выход AMUX HI	*Выход AMUX LO*	*Интерпретация*	*Мнемоника/AMUX*	
D(HI)	D(LO)	D unchanged	DIRECT	0
D(LO)	D(LO)	D(LO) duplicated	DLO#DLO	1
D(C) replicated	D(LO)	D(LO) sign extended if D(C)=ALU 07	SIGNEXT	2
COUNTER	D(LO)	Counter introduced into datapath	COUNTR	3
D(HI)	D(HI)	D(HI) duplicated	DHI#DHI.	4
D(LO)	D(HI)	Byte swap	SWAB	5
D(C)	D(HI)	Right shift 8 of D(HI)	RIGHT-8	6
COUNTER	D(HI)	Counter introduced into datapath	COUNTR#DHI	7

2. BMUX. Выход AMUX может проходить прямо через BMUX (DIRECT, BMUX = 0) или может быть сдвинут вправо на 4 бита с одновременным копированием D(C) в старшие 4 бита (RIGNT-4, BMUX = 1).

3. CMUX. 2-битовое поле CMUX осуществляет один из следующих четырех выходов:

Выход CMUX	*Интерпретация*	*Мнемоника/CMUX*	
BMUX left one, SENDMUX enters low bit	Arithmetic left shift by 1	LEFT-1	0
BMUX	BMUX unchanged	DIRECT	1
BMUX right one with D(C) into high bit	BMUX right shifted by 1	RIGHT-1	2
BMUX right two with D(C) into high bits	BMUX right shifted by 2	RIGHT-2	3

Таким образом, корректный подбор D(C) и соответствующий выбор сдвигов вправо (0 или 8 *плюс* 0 или 4, *плюс* — 1 или 0, либо 1 или 2) позволит в одном микроцикле произвести сдвиг вправо с учетом (арифметический сдвиг) или без учета знака (логический сдвиг). Схема позволяет производить сдвиг на $n = 0—14$ бит. 16-битовый результат направляется на шину A.

В следующем примере *одна* микрокоманда выполняет следующие операции: выборка содержимого регистра D, сдвиг его вправо на 11 разрядов (8 + 4 — 1 = 11), вычитание из результата содержимого регистра R0 и запоминание в D окончательного результата.

Микрокоманда:

```
ALU/A-MINUS-B,COUT/ALU15,
BEN/BSPLO,BSEL/IMMED0,RIF/4,
AEN/CMUX,AMUX/RIGHT-8,BMUX/RIGHT-4,ASEL/LEFT-1,
CLKD/YES,WHEN/P2-T
```

Среди других полезных операций, реализуемых посредством дерева сдвига, можно назвать перестановку байта и расширение

младшего байта регистра D. Первая представлена обычной командой PDP-11, а вторая выполняется, если в качестве операнда-приемника в команде MOVB взят *gp*-регистр.

6.4.2. Регистр сдвига

16-битовый регистр сдвига (SR) может дополнительно расширяться на 4 бита за счет регистра GD. GD начнет действовать, если требуется производить тактирование регистра SR (разд. 6.6).

При сдвиге вправо содержимое SR смещается вправо на единицу. Одновременно младший бит BMUX занимает место старшего в SR, а младший бит SR сдвигается на место старшего в GD, если для GD разрешено тактирование.

При сдвиге влево SR смещается влево на единицу. Одновременно старший бит SR выталкивается в SENDMUX, а старший бит GD *или* D(C) попадает в младший бит SR.

Работа SR, когда он тактируется, определяется двумя битами RES (разд. 6.6), т. е. нет сдвига, сдвиг влево, сдвиг вправо или прием вывода ALU. В целом комбинация, состоящая из регистров D и SR, позволяет сдвигать 32-битовые числа. Например, когда AMUX передает D прямо через BMUX ($AMUX = 0$, $BMUX = 0$) и CMUX выполняет сдвиг влево на 1 ($CMUX = 0$), то D будет смещен влево на единицу, старший бит SR попадет в SENDMUX (если SR тактируется) и далее — в младший бит регистра D. Если старшее слово 32-битового числа было предварительно помещено в D, а младшее — в SR и если RES специфицирует сдвиг влево в регистре SR, то произойдет сдвиг на один бит влево в 32 битах, помещенных в D и SR. Точно так же, если $AMUX = 0$, $BMUX = 0$ и $CMUX = 1$ (сдвиг вправо), то младший бит BMUX (т. е. младший бит D) займет место старшего в SR, в то время как SR сдвигается на один бит вправо и, в конце концов, происходит сдвиг вправо на один бит в 32-битовом числе, находящемся в D и SR. Таким образом выполняются как арифметические, так и логические сдвиговые операции, в зависимости от выбора D(C).

В разд. 6.8 мы покажем, как, используя в комбинации D и SR, осуществить умножение двух 16-битовых слов, получая 32-битовый результат.

6.4.3. Дерево сдвига и регистр сдвига

С помощью дерева сдвига (ST) можно выполнить многократный сдвиг вправо регистра D за один микроцикл. При этом младшие биты выпадают вообще и получается 16-битовый результат. С другой стороны, дерево сдвига обеспечивает сдвиг в регистре D на один бит влево. Комбинация D, SR позволяет производить сдвиг влево на один бит или сдвиги вправо 32-битового числа в одном микроцикле.

6.5. МАГИСТРАЛЬ ВВОДА-ВЫВОДА

Описанные ранее компоненты магистрали данных выполняют арифметические/логические операции над данными, расположенными в быстрой памяти A, B или C и в регистрах D или SR. Как же данные поступают в магистраль данных из основной памяти? Каким образом микропрограмма получает доступ к данным, находящимся в основной памяти? Где находятся MAR и MDR, которые описаны в параграфах 2.5.1 и 3.5.4?

На эти вопросы можно дать следующий ответ: если микрополе *CLKBA* установлено в 1, тогда 16-битовый *виртуальный* адрес требуемых данных будет загружен в регистр BA из специфицированного источника шины A в момент P1. Если DATA-вход инициирован, то данные, адрес которых указывается содержимым регистра BA, будут выбраны из основной или кэш-памяти и загружены в CSP в момент P3 *следующего* микроцикла (если поле *WRCSP* равно единице). По соглашению, соответствующие микрополя устанавливаются так, что CSP [15] рассматривается как MDR и принимает вводимые данные. Заметим, что при отсутствии требуемых данных в кэш-памяти (т. е. они в основной памяти) автоматически генерируется сигнал «подавить импульс», который прекращает выдачу синхронизирующих импульсов до тех пор, пока не закончится цикл загрузки UNIBUS. Вводимые данные одновременно могут быть загружены в IR, если установлены биты соответствующих микрополей управляющего слова.

Точно так же, если данные должны выводиться в основную память (из регистра D), то обновление кэш-памяти и CLKBA происходит в момент P3 этого же микроцикла, а P3 следующего микроцикла подавляется до тех пор, пока не закончится обновление памяти.

Таким образом, в качестве регистра MAR в 11/60 служит регистр BA, в то время как CSP [15] используется как MDR.

6.6. КОНЦЕПЦИЯ ОСТАТОЧНОГО УПРАВЛЕНИЯ

В течение микроциклов часть информации остается неизменяемой, или статической. Так, после сдвига вправо 32-битового числа, выполняемого с помощью SR и ST совместно с D, SR остается в режиме сдвига вправо, а D(C) *всегда* равен либо 0 (логический сдвиг), либо знаковому биту ALU (арифметический сдвиг). Регистр SR управляется содержимым регистра RES, которое изменяется в том случае, если в последующих микроциклах операции, выполняемые регистрами SR и GD, будут отличаться от ранее выполняемых. Поэтому регистр RES называется регистром «остаточного» управления. И вообще управление сигналами, которые статистически не должны изменяться довольно часто, происходит «косвенно», через посредство регистров, а не бит микрополей.

6.7. ПОСЛЕДОВАТЕЛЬНОСТИ МИКРОКОМАНД И МИКРОТЕСТЫ ПЕРЕХОДОВ

При отсутствии переходов команды PDP-11 выполняются последовательно без каких-либо специальных указаний. В противоположность этому каждая *микрокоманда* PDP-11/60 в 5-битовом поле-микроуказателе (*UPF*) содержит адрес следующей выполняемой микрокоманды. При изменении содержимого *UPF* в процессе обработки микрокоманды осуществляются переходы, нарушающие последовательное выполнение микрокоманд. Так как 11/60 может допускать частично перекрывающую загрузку (т. е. загрузка очередной микрокоманды может начаться в период выполнения текущей, см. разд. 6.9), возможны ошибки, если не предусмотреть программные методы проверки, используемые в тех случаях, когда результат текущей микрокоманды диктует необходимость осуществления перехода.

6.7.1. Микропереходы

5-битовое поле (*UBF*) управления микропереходами управляет мультиплексором переходов (BUTMUX), *выход* которого объединяется логической операцией OR с UPF, образуя следующий микроадрес (NUA). Входной информацией для BUTMUX являются различные состояния машины. Так, например, когда на выходе BUTMUX все нули, то NUA = UPF. Если в поле *UPF* все биты — единицы, то после логической операции OR в поле NUA все биты также будут установлены в единицу, поэтому эта часть схемы определяет длину участка, в котором указывается новый микроадрес и на который может воздействовать BUTMUX (рис. 6.4).

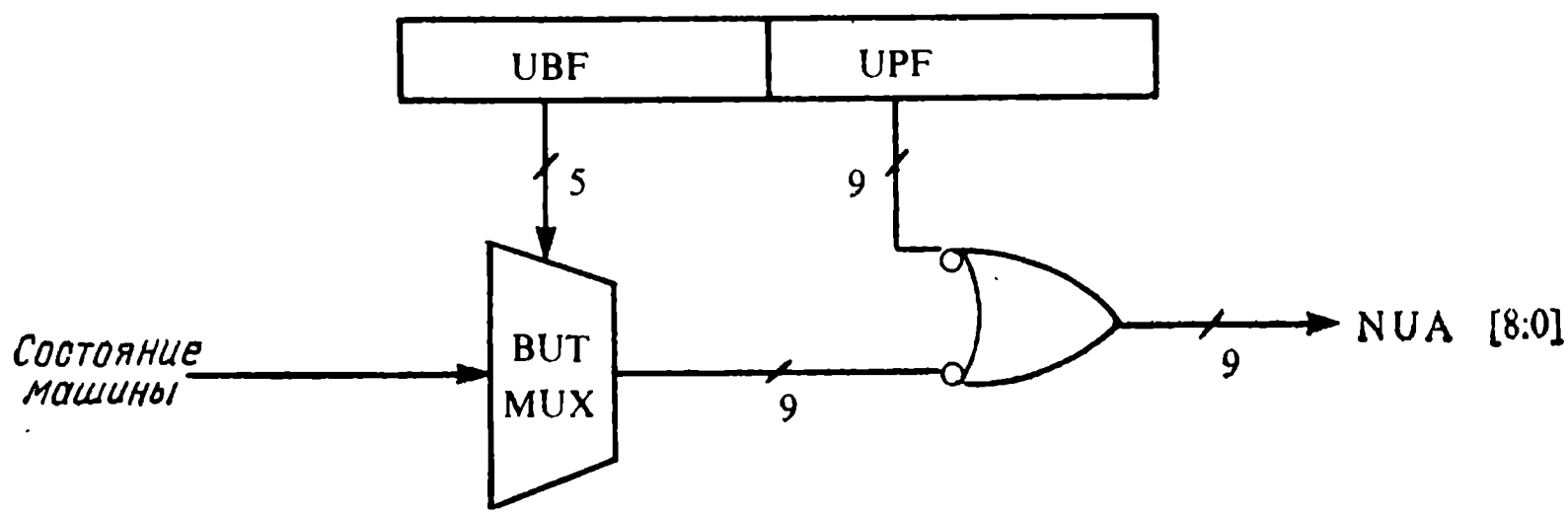


Рис. 6.4. Формирование следующего микроадреса (NUA) на основе UBF, BUTMUX и UPF

Каждый микротест перехода (BUT) связан с одним из 32 возможных значений поля *UBF*. Несколько микротестов (активные BUT) вызывают определенные изменения состояний машины или воздействуют на NUA. Формирование NUA для следующей микрокоманды начинается в начале выполнения текущей; аналогично загрузка следующей микрокоманды начинается до завершения текущей. При этом предполагается, что условия для действия микротеста BUT на условный переход (т. е. на формирование NUA)

должны быть установлены в микроцикле, *предшествующем* микроциклу, в котором формируется NUA. Итак, условия для перехода формируются в микроцикле 1, тестирование и формирование NUA с помощью кодов BUT происходят в микроцикле 2, а в течение микроцикла 3 будет выполнена та микрокоманда, на которую будет указывать результирующий условный микропереход.

6.7.2. Обычные микротесты

Обычные микротесты тестируют определенные поля, чтобы потом иметь возможность воздействовать на несколько вариантов переходов. И снова, если 1 в *UPF* остается неизменной при выполнении логической операции OR с некоторым другим полем, то микропрограммист может уменьшить количество переходов, заранее избрав микроадрес перехода. Ниже приводится несколько обычных BUT.

1. [*UBF* = 0] BUT(CASE) потенциально может обеспечить 16 вариантов одного и того же перехода, используя для этого 4 младших значащих бита SP-регистра как выход BUTMUX.

2. [*UBF* = 1] BUT(DOP) применяется для декодирования двухоперандных команд PDP-11 и потенциально обеспечивает 16-вариантный переход с помощью 4 самых старших значащих бита регистра IR как выхода BUTMUX.

3. [*UBF* = 3] BUT(SOP) служит для декодирования однооперандных команд PDP-11, потенциально обеспечивает 16-вариантный переход на основе содержимого бит от 6 до 9 регистра IR как выхода BUTMUX.

4. [*UBF* = 12] BUT(GUARD) употребляется при проверках операций сдвигов, потенциально обеспечивая 4-вариантный переход, используя для этого биты 2 и 3 регистра GUARD, как выход BUTMUX.

Существует еще ряд микротестов. Некоторые из них будут анализироваться позже в примерах.

6.7.3. Активные микротесты. Счетчик

Активные BUT воздействуют также на выполнение логической операции OR между выходом BUTMUX и полем *UPF*, допуская *n*-вариантные переходы; однако такие BUT вызывают некоторые изменения в состоянии машины. Наибольший интерес, с этой точки зрения, представляет для нас микротест BUT(COUNT).

Присутствие счетчика в магистрали данных PDP-11/60 дает возможность микропрограммисту существенно усилить выполнение каждой микрокоманды действием, подобным действию команды SOB. Делается это так. 8-битовое отрицательное число загружается в счетчик; затем каждый раз, когда специфицирован BUT(COUNT), счетчик проверяется и увеличивается. Все единицы, оказавшиеся на момент проверки в счетчике, объединяются логической операцией OR с LSB поля *UPF*, создавая возмож-

ность 2-вариантного микроперехода. Как мы увидим, такие операции дают мощную и эффективную по времени выполнения конструкцию, которая позволяет отдельной микрокоманде делать переход на саму себя. В свою очередь, это используется в повышении эффективности выполнения операции умножения и различного рода многобитовых сдвигов 32-битового слова.

6.8. ПАРАЛЛЕЛИЗМ В МИКРОКОДИРОВАНИИ: КОМАНДА MUL

Как было отмечено раньше (параграф 5.2.4), числа со знаком не могут быть перемножены прямо, посредством простого алгоритма сдвиг/сложение. Поэтому сначала мы рассмотрим суть вычислительного ядра команды MUL, когда она выполняется с помощью регистров D и SR.

6.8.1. MUL с использованием D-регистра, регистра сдвига и дерева сдвига

Предположим, что две величины *без знака* — множимое (MAND) и множитель (MIER) — находятся на хранении в регистре временного хранения R(T) и в SR соответственно. Далее мы предположим, что счетчик загружен отрицательным числом —16, а в регистре RES установлен код выполнения сдвига вправо в SR, когда он будет тактироваться. Рис. 6.5 показывает выполнение микрокоманды в течение 16-шагового внутреннего цикла. На каждом его шаге *текущая* микрокоманда вызывает сдвиг или сдвиг/сложение, как это предписывается тестом, осуществлявшим тестирование бита *предыдущей* микрокоманды.

И так как каждая микроинструкция, выполняющаяся во внутреннем цикле, делает условное 16-битовое сложение, сдвиг вправо на 1 бит 32-битового целого, тестирование на 1 или 0 младшего бита 32-битового целого, условный переход в зависимости от результата, а также настройку 8-битового счетчика на условный переход, в зависимости от результата, то все это наглядно демонстрирует параллелизм, внутренне присущий микропрограммному управлению мощной микромашиной.

Последовательность микрокоманд на рис. 6.5 может быть пояснена следующим образом.

1. Первая микрокоманда обнуляет регистр D и осуществляет условный переход на одну из пары макрокоманд по результату тестирования младшего бита MIER, находящегося в SR. *Переход на шаг 2.*

2. Следующая *пара* микрокоманд — первая во внутреннем цикле — выполняет операции D ← D + R(T) или D ← D + 0. То, какая из микрокоманд данной пары будет работать, зависит от младшего бита MIER, проверяемого в первой микрокоманде. SR сдвигается на один бит вправо, а в старший бит SR засылается нуль. Если после увеличения счетчика (COUNT) он остается меньше 377, что и должно быть в нашем случае, то произойдет условный переход

на следующую пару микрокоманд. Переход будет зависеть от младшего бита SR, т. е. 2-го младшего значащего бита множителя MIER. *Переход на шаг 3.*

Рис. 6.5. Микрокоманды, выполняющие команду MUL

3. *Вторая пара* микрокоманд выполняет операции $D \leftarrow D/2 + R(T)$ или $D \leftarrow D/2 + 0$. Выбор одной из них определяется, как и выше, значением младшего бита SR. SR сдвигается вправо на 1. В старший бит SR войдет «выпавший» бит из регистра D. Затем выполнится условный переход по младшему биту MIER, с передачей управления одной из микрокоманд *второй* (COUNT $<$ 377) (*переход на шаг 3*) или третьей пары (COUNT $=$ 377) (*переход на шаг 4*). При этом микрокоманда из второй пары будет выпол-

няться в течение каждых 15 циклов, необходимых для того, чтобы счетчик достиг числа 377.

4. *Третья пара* микрокоманд (не показана на рисунке) подобна второй паре. Она выполняет операции D ← D/2 + R(T) или D ← D/2 + 0, в зависимости от результата проверки младшего бита SR, сделанной в предыдущей микрокоманде, SR также сдвигается вправо на один, а в старший бит SR из регистра D войдет его «выпавший» бит. Эта пара выполнит последнюю операцию сдвиг/условное сложение внутреннего цикла команды MUL. Управление будет передано следующей микрокоманде в последовательности MUL — первой микрокоманде завершающей фазы.

Итак, мы только что увидели, что параллелизм, внутренне присущий магистрали данных машины, подобной 11/60, дает возможность писать мощные коды, однако забота о правильности этого кодирования остается на программисте.

6.8.2. MUL с использованием D-регистра и дерева сдвига

В параграфе 6.4.1 мы заметили, что в одном микроцикле можно выполнить сдвиг вправо на 14 позиций в регистре D, добавить или вычесть из результата число, находящееся в быстрой памяти B *и* полученное 16-битовое значение снова поместить в D. Это означает, что, применяя алгоритм Бута, приведенный в параграфе 5.2.4, мы можем выполнить 16-битовое умножение в среднем за 5,33 микроцикла при условии, что у нас написаны обычные микроподпрограммы для каждого нужного множителя. В итоге устраняется необходимость применения специальных приемов микропрограммирования вообще, однако предполагается, что использованы всего лишь несколько различных множителей и что со временем они не изменяются (или изменяются нечасто). Многие алгоритмы обработки цифровых сигналов фактически удовлетворяют этим требованиям. Предлагаем читателю обсудить приложение микропрограммирования к программному обеспечению DSP.

6.9. ЗАКЛЮЧЕНИЕ ПО ГЛАВЕ

Мы провели анализ ресурсов, предоставляемых 11/60 на микромашинном уровне, и дали несколько примеров, показывающих, как применять их для выполнения различных операций, включая фрагменты инструкций PDP-11. Мы не исследовали возможности микроподпрограмм 11/60 или всю изощренность различных микротестов.

Однако ясно, что микропрограммы сложнее программ, написанных на макро- или машинном языке, прежде всего потому, что «микропрограммист» должен следить за всеми многочисленными событиями, происходящими в одной микрокоманде. А для получения максимума скорости необходимо реализовывать перекрывающееся выполнение микрокоманд. В гл. 9 мы покажем, как микропрограммирование используется для увеличения скорости. И еще

одна область, где применяется микропрограммирование,— это эмуляция ЭВМ. Написав соответствующую микропрограмму, мы можем заставить 11/60 работать так, как если бы эта ЭВМ выполняла программу *любой желаемой машины.* Так, например, на PDP-11/60 можно микропрограммно эмулировать ЭВМ PDP-8.

УПРАЖНЕНИЯ

1. С помощью простейшей ЭВМ (рис. 6.1) запишите микропоследовательность для одноадресной команды (LDA), имеющей формат | LDA | адрес | и выполняющей загрузку аккумулятора (из памяти).

2. Сравните и противопоставьте микропоследовательности в управляющей памяти командам в основной памяти. Какая существует разница (если существует) между микропрограммированием и программированием?

3. В VAX-11/780 около половины микрокодов, обеспечивающих выполнение команд, резидентны в памяти, защищенной от записи (ROM), в то время как другая половина считывается в память, допускающую запись (WCS), с гибкого диска каждый раз, как включается машина. Каковы преимущества такой схемы? Почему невыгодно хранить на диске все микрокоды?

4. Объясните, насколько в 11/60 удобен способ спецификации *следующего микроадреса* (NUA) с точки зрения коррекции ошибок микрокода.

БИБЛИОГРАФИЯ

С целью разностороннего подхода к пониманию основ вычислительной техники рекомендуется внимательно прочитать более ранние книги: Gear C. W. (1974), Stone H. S. (1975) и Abrams M. D. и Stein P. G. (1973). Можно рекомендовать также специальные книги по логическому проектированию Phister M. (1959), архитектуре компьютера Foster C. C. (1970), микропрограммированию Husson S. S. (1970) и системным принципам Hellerman H. (1973), если читатель желает углубить свои знания по указанным вопросам. Секции по микропрограммированию основаны на работе Jackson T. L. (1977). Работа более детально освещает вопросы микропрограммирования по сравнению с обзором, данным в этой книге.

Глава 7

ПРОГРАММИРОВАНИЕ ВВОДА-ВЫВОДА

Выполнение в ЭВМ программы вычислений не обеспечивает возможности решить даже простую практическую задачу, поскольку в любом случае необходимо наличие путей ввода данных в машину и вывода результатов. Следовательно, программисту должны быть предоставлены средства передачи информации между ЭВМ и периферийными устройствами, которые осуществляют ввод или служат для вывода.

Для выполнения функций ввода-вывода (I/O) программист обязан специфицировать данные, а также те устройства, с которых они должны вводиться или на которые выводиться. Кроме того, требуется указание режимов управления этими устройствами. В зависимости от модели ЭВМ, функции I/O сопровождаются остановом CPU до завершения операции обмена либо допускают одновременную работу процессора и каналов I/O. Если CPU останавливается при выполнении операции I/O, то мы говорим, что операции I/O и CPU *взаимно блокируют* друг друга. Если же оба процесса могут выполняться одновременно, мы говорим, что I/O и вычисления *совмещаются.*

Совмещенные операции становятся стандартным режимом для большинства систем. Этот режим имеет несколько форм. В одной из них функция I/O оперирует одним словом данных. При этом в течение операции слово удерживается в специальном регистре, например в аккумуляторе.

В другой форме ввод-вывод выполняется прямо между памятью и устройством. Такой режим работы требует специального оборудования, реализующего так называемый *прямой доступ* (DMA), при котором функции I/O выполняются с минимальной зависимостью от CPU.

При третьем режиме обмена информация передается большими блоками между памятью и устройством I/O. Для обеспечения *передачи блоков* существуют специальные регистры. В них хранятся количество слов, которые надо еще передать, номер текущего устройства I/O и адрес слова данных в памяти, а также сами данные. Будучи инициирована инструкцией I/O, передача блоков происходит совместно и независимо от работы CPU до окончания передачи (т. е. пока счетчик слов не будет равен нулю или не произойдет ошибки адресации).

Наличие магистрали данных DMA, а также подобной магистрали для доступа к памяти CPU иногда может привести к

конфликту. Чтобы этого не случилось, запросам на I/O по магистрали DMA отдается предпочтение по отношению к процессору. Это объясняется тем, что запросы к памяти, обусловленные функциями I/O, зависят от времени, возникают относительно редко и являются непродолжительными по времени выполнения. Такой подход называется «кражей цикла», ибо циклы доступа к памяти используются устройством I/O за счет CPU.

Из этого краткого введения должно быть совершенно ясно, что различные возможности I/O в большой степени зависят от машины. Сложность системы ввода-вывода в конечном счете определяет сложность программирования операций обмена данными. На PDP-11 программирование устройств I/O чрезвычайно простое и не требует дополнительных команд.

Это обусловлено особенностями шины UNIBUS, описанной в гл. 3. UNIBUS обеспечивает унифицированную структуру адресации, в которой осуществляется прямое обращение к регистрам управления, состояния и данных *периферийных устройств,* как к обычным ячейкам памяти. Поэтому операции с этими регистрами, такие, как обмен и манипуляция в них данными, исполняются с помощью обычных команд обращения к памяти.

7.1. ОСНОВЫ ПРОГРАММИРОВАНИЯ ОПЕРАЦИЙ ВВОДА-ВЫВОДА И ИХ ВЫПОЛНЕНИЕ

Использование команд обращения к памяти для работы с регистрами периферийных устройств существенно увеличивает гибкость программирования I/O. Например, информация в регистре устройства I/O может прямо сравниться с некоторой величиной и таким образом определить необходимость перехода:

```
CMPB    @#TKB,#'Y        ;IS CHARACTER = Y?
BEQ     YES              ;BRANCH IF SO
```

Здесь программа следит за появлением символа «Y» в буфере данных клавиатуры (TKB) и выполняет переход, если этот символ обнаружен. Как видим, при этом нет необходимости пересылать информацию для сравнения в промежуточный регистр.

Если интересующий нас символ появился и мы хотим его сохранить, то это можно сделать с помощью команды обращения к памяти и передать его в буфер пользователя или другому периферийному устройству. Команда

```
MOVB    @#PRB,LOC        ;SAVE CHARACTER IN MEMORY "LOC
```

пересылает символ из буфера устройства чтения с перфоленты (PRB) в ячейку памяти, определенную пользователем.

Еще один аспект программирования заключается в том, что с помощью регистра периферийного устройства ввода или вывода можно выполнять арифметические операции. Следовательно, нет

необходимости использования общих регистров или другой памяти для передачи данных, выполнения арифметических операций и операций сравнения. Вместо этого регистр периферийного устройства может служить аккумулятором.

7.1.1. Регистры периферийных устройств

Все периферийные устройства специфицируются множеством регистров, у которых адресация такая же, как и у основной памяти, а возможность манипуляции данными — как у аккумуляторов. С каждым устройством связаны два типа регистров:

1) регистры управления и состояния;
2) регистры данных.

Любое периферийное устройство имеет один или более регистров управления и состояния (CSR), содержащих всю необходимую информацию для связи с устройством. В таблице ниже показан общий формат содержимого CSR. Этот формат соблюдается не для всех устройств, но полезен для понимания вопроса. Одни устройства для отражения состояния требуют менее 16 бит, в то время как другим для этого нужны дополнительные регистры управления и состояния. В таблице описано назначение бит, *обычно* используемых в регистрах CSR периферийных устройств.

Каждое устройство имеет, кроме CSR, по крайней мере один буферный регистр для временного хранения данных, передаваемых в машину или из нее. Число и тип регистров данных зависят от функций устройства. Устройство чтения и вывода на перфоленту использует 8-битовые буферные регистры. В диске задействован 16-битовый регистр данных, а некоторые устройства для буферизации данных могут применять два 16-битовых регистра.

7.2. БАЗОВЫЕ УСТРОЙСТВА

Наиболее часто к PDP-11 присоединяются следующие базовые устройства: терминал (например, устройство печати с клавиатурой DECwriter или телетайп ASR-33), высокоскоростное устройство чтения/вывода на перфоленту (например, DEC PC-11) и большая внешняя магнитная память для хранения информации (например, диски RK05, RK06/07 или RL01 DECpack, гибкие диски RX01 или магнитная лента DECtape). В комплексе ASR-33 имеется устройство чтения с клавиатуры/перфоленты и вывода на печать/перфоленту. Работа ASR-33 будет описана ниже, но здесь отметим, что каждый терминал, соединенный с консольным интерфейсом компьютера, позволяет осуществлять ввод с клавиатуры или отображать выводимую информацию.

7.2.1. Терминальная клавиатура/перфоввод

Устройство управления терминалом включает 8-битовый буфер (ТКВ), в котором преобразуется из последовательного кода и

хранится последний символ, введенный с клавиатуры или прочитанный с ленты. В зависимости от команды в программе, содержимое ТКВ может быть передано в общий регистр или в ячейку памяти.

Назначение бит регистров периферийных устройств

Бит	Назначение	Описание
15—12	Ошибки	Обычно каждую ошибку специфицирует отдельный бит. Если же требуется бо́льшая длина поля ошибок, то она может быть достигнута за счет данного либо дополнительного слова состояния. Обычно бит 15 равен логическому сложению всех битов ошибок.
11	Занято (BUSY)	Индицирует, что устройство выполняет операцию.
10—8	Номер устройства	Часто один контроллер управляет более чем одним устройством. Например, дисковая система имеет более чем один дисковод, а аналого-цифровой преобразователь более чем один канал.
7	Сделано или Готов (DONE или READY)	Регистр может иметь бит DONE или бит READY, или пару бит DONE — BUSY. Эти биты очищаются и устанавливаются аппаратно, но могут опрашиваться из программы с целью определения готовности устройства. Бит DONE можно рассматривать как знаковый. Команда тестирования младшего байта регистра очищает бит N, если бит 7 есть нуль, и устанавливает бит N, если бит 7 есть 1.
6	Прерывание разрешено (Interrupt Enable)	Если бит 6 установлен, то будут происходить прерывания после выполнения функции или возникновения ошибки.
5—4	Биты расширения памяти	Позволяют устройству использовать полностью все 18 бит для спецификации адреса на UNIBUS.
3—1	Функция устройства	Указывает операции, которые должно выполнять устройство. Например, устройству чтения с ПЛ может быть задано «прочитать символ», а диску «прочитать блок слов из памяти и записать их на диск».
0	Разрешение (Enable)	Если установлен, то это означает, что устройство готово выполнять свои функции.

Когда на таком терминале нажимается клавиша, то соответствующий символ принимается в ТКВ и после окончания операции выставляется бит DONE. Если происходит чтение с низкоскоростного телетайпа ASR-33, то бит готовности чтения (RDR ENB) устанавливается в 1. В свою очередь, это вызывает установку в 1 бита занятости (BUSY). Как только все готово для ввода символа, управление обесточивает реле телетайпа, чтобы освободить защелку ввода ленты. Теперь остановка произойдет после того,

как символ будет полностью прочитан и опознан в буфере (ТКВ). Затем бит занятости очищается и устанавливается флаг «сделано» (DONE). Символ из ТКВ должен быть прочитан в течение 18 миллисекунд после установки DONE, иначе может произойти потеря информации.

Регистр состояния (TKS) терминала клавиатура/устройство чтения (только ASR-33)

Бит	Назначение	Описание
15—12		Не используется.
11	Занято (BUSY)	Индицирует, что терминалу передаются биты информации или бит начала его работы. Только чтение.
10—8		Не используются.
7	Сделано (DONE)	Устанавливается, когда символ находится в буфере. Очищается, как только появляется ссылка на буфер (т. е. данные выбираются из буфера). Вызывает прерывание, если бит 6 установлен. Только чтение.
6	Разрешены прерывания устройству чтения	Разрешается прерывание, если установлен бит 7.
5—1		Не используются.
0	Разрешение устройству чтения с ПЛ	Устройству чтения с ПЛ (для ASR-33) разрешено читать один символ. Только запись. Очищается во время операции чтения.

Примечание. Биты 11, 7, 6 и 0 очищаются также по импульсу INIT, выдаваемому в момент включения машины или когда программистом выдается команда «RESET».

Буфер (ТКВ) терминала клавиатура/устройство чтения

Бит	Назначение	Описание
15—8		Не используется.
7—0	Данные	Содержит прочитанный символ. Только чтение.

Любая ссылка к ТКВ (как к слову или байту) или ТКВ + 1 очищает DONE. Биты «не используется» и «только запись» читаются всегда как нули. Загрузка бит «не используется» или «только чтение» не оказывает влияния на состояние CSR. Мнемоника INIT означает, что инициализирующие сигналы могли быть выданы командами ON, POWER UP, командой START с консоли или командой RESET.

7.2.2. Терминальная печать/перфовывод

По команде из программы символ пересылается из памяти (или из общего регистра) в ТРВ для последующей передачи на

устройство вывода на перфоленту/печать. Передача этой информации производится с обычной для телетайпа скоростью и требует на выполнение 100 миллисекунд. Флаг READY устройства вывода на перфоленту/печать индицирует готовность TPB принять новый символ. В режиме обслуживания при подключенном выходе TPB ко входу TKB возможна проверка функционирования выводных линий терминала.

Регистр состояния (TPS) терминала печать/устройство вывода на перфоленту (только ASR-33)

Бит	Назначение	Описание
15—8		Не используются.
7	Готово (READY)	Устройство в готовности. Устанавливается по команде INIT, очищается, когда в буфер поступила информация, и снова устанавливается, когда операция печать/вывод на ПЛ закончена. Вызывает прерывание, если бит 6 = 1. Только чтение.
6	Прерывание разрешено	Разрешается прерывание, если бит 7 установлен.
5—3		Не используются.
2		Функция обслуживания,
1—0		Не используются.

Буферный регистр (TPB) терминала печать/устройство вывода на ПЛ

Бит	Назначение	Описание
15—8		Не используются.
7—0	Данные	Хранит символ, который должен быть напечатан или выведен на ПЛ. Только запись.

Любая команда, которая модифицирует TPB, очищает флаг READY и инициирует вывод перфоленты.

Как уже упоминалось, с терминалом связаны четыре адресуемых регистра. По любой команде PDP-11 можно читать или записывать информацию в них. Они обычно имеют следующие адреса:

Регистр	*Адрес*
TKS	177560
TKB	177562
TPS	177564
TPB	177566

При использовании MACRO-11 можно обращаться к регистрам устройств по их символьным именам. Для этого символам нужно присвоить соответствующий адрес (например, TKB = 177562).

7.2.3. Простой пример программирования

Раз клавиатура телетайпа отделена от устройства печати, есть возможность вывода на печать каждого символа, введенного с клавиатуры, который как «эхо» будет отображаться на устройстве печати. Образец программы для этой цели приводится ниже.

```
        TKS=177560              ;DEFINE
        TKB=TKS+2               ;STATUS
        TPS=TKS+4               ;AND BUFFER
        TPB=TKS+6               ;REGISTERS
ECHO:   INC     @#TKS           ;SET PAPER TAPE READER TO READ
LOOP1:  TSTB    @#TKS           ;IS CHARACTER IN BUFFER YET?
        BPL     LOOP1           ;BRANCH IF NOT
LOOP2:  TSTB    @#TPS           ;IS PRINTER FREE?
        BPL     LOOP2           ;BRANCH IF NOT
        MOVB    @#TKB,@#TPB     ;READER BUFFER TO PRINTER
        BR      ECHO            ;LOOP AGAIN
```

Здесь наглядно демонстрируется значение байтовой работы с регистром. Если не использовать бит DONE как знаковый бит, то пришлось бы формировать слово-маску и производить тестирование бита с последующим переходом по нулевому значению. И хотя такое альтернативное решение возможно, оно все же не так привлекательно (с точки зрения эффективности по времени выполнения *и* занимаемому объему памяти), как команда TSTB. В то же время возможность применения такой техники программирования еще раз наглядно иллюстрирует взаимное влияние оборудования и программного обеспечения при создании компьютера.

Установку бита готовности чтения в этой программе можно было бы и не делать. Однако, включая команду, которая осуществляет это, мы делаем нашу программу инвариантной к устройству, поскольку теперь она может принимать данные как с клавиатуры, так и с устройства чтения с перфоленты. Подобно этому, вывод теперь может производиться как на печать, так и на перфоленту. Все, что нужно сделать пользователю, это поместить перфоленту в устройство чтения и установить на нем режим готовности или включить устройство вывода на перфоленту. Тогда эти перфоленточные устройства будут действовать параллельно с другим оборудованием терминала (например, клавиатурой или устройством печати). Следовательно, одна эта программа воспринимает работу любой допустимой комбинации компонентов телетайпа.

7.2.4. Программа восьмеричного дампа

Программисты, применяющие язык Ассемблера, часто пользуются таким средством программирования, как программа дампа памяти. Эта программа оказывается весьма полезной при разработке или отладке программы, так как предоставляет программисту копию такой программы или ее части, находящейся в памяти.

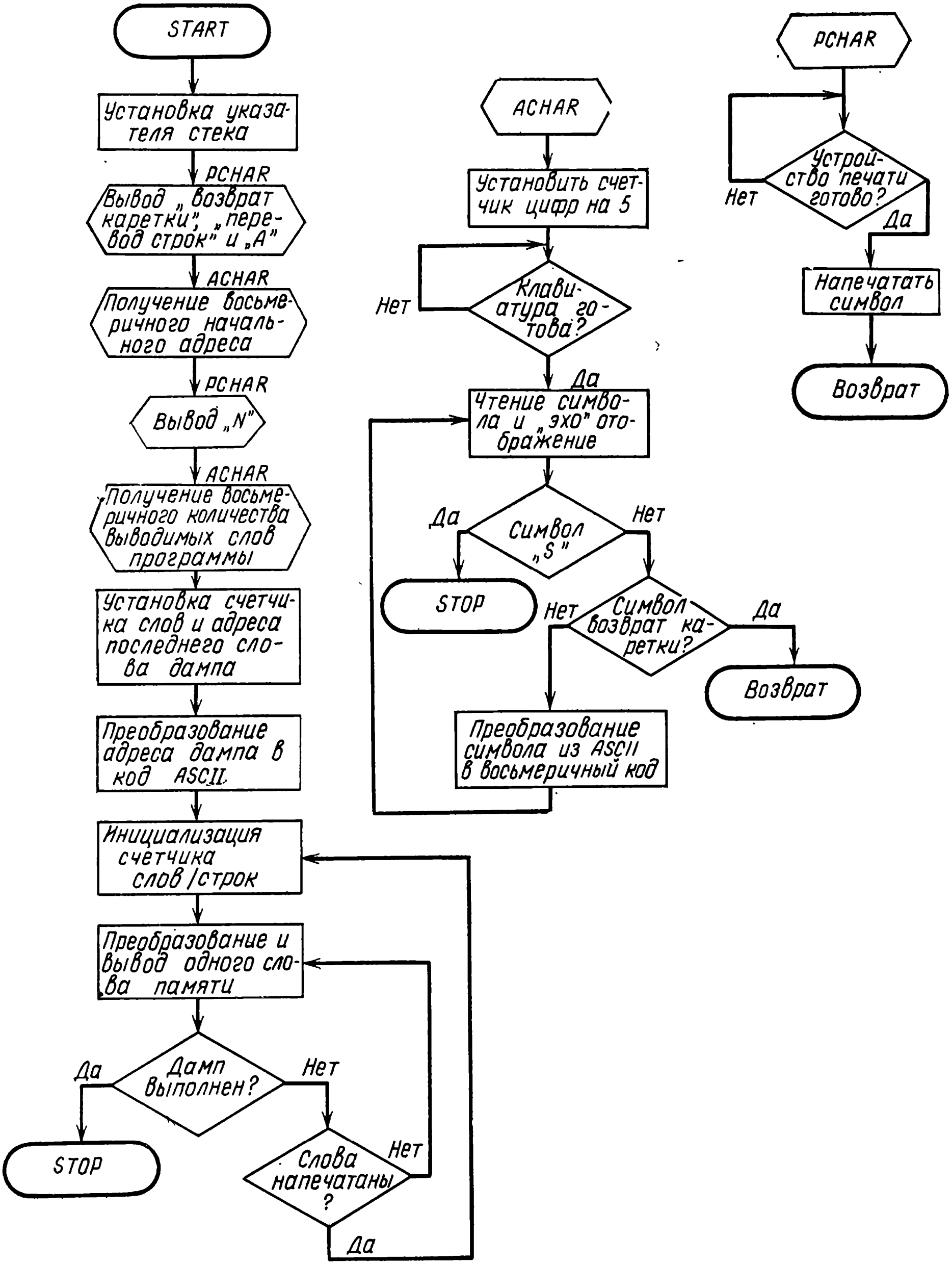


Рис. 7.1. Блок-схема программы восьмеричного дампа

```
;              EDUMP - AN OCTAL DUMP PROGRAM WRITTEN IN PIC
;
; INPUTS ARE N -- THE NUMBER OF WORDS;  AND A -- THE STARTING ADDRESS
;
; OUTPUT IS THE STARTING MEMORY ADDRESS
;     AND THE CONTENTS OF UP TO 8 WORDS OF MEMORY, PER LINE OF OUTPUT
;
        TKS=177560
        TKB=TKS+2
        TPS=TKS+4
        TPB=TKS+6
;
        CR=15
        LF=12
;
CORE:   MOV     PC,SP           ;SET UP STACK POINTER USING
        TST     -(SP)           ; POSITION INDEPENDENT CODE (PIC)
        CLR     @#TKS           ;INITIALIZE KEYBOARD STATUS REGISTER
        CLR     @#TPS           ;INITIALIZE PRINTER STATUS REGISTER
        MOVB    #CR,R0          ;PRINT INITIAL CARRIAGE
        JSR     PC,PCHAR        ;RETURN AND
        MOVB    #LF,R0          ;LINE FEED USING
        JSR     PC,PCHAR        ;PUT CHARACTER SUBROUTINE
ADDR:   MOVB    #'A,R0          ;PRINT AN "A"
        JSR     PC,PCHAR
        JSR     PC,ACHAR        ;ACCEPT UP TO 5 OCTAL DIGITS AS ADDRESS
        MOV     R5,R1           ;R1 CONTAINS START ADDRESS
        MOVB    #'N,R0          ;PRINT AN N FOR NUMBER OF WORDS
        JSR     PC,PCHAR
        JSR     PC,ACHAR        ;ACCEPT ≤ 5 OCTAL DIGITS
        MOV     R5,R2           ;FORM WORD COUNT NUMBER
        ADD     R5,R2           ;TO BE DUMPED
        ADD     R1,R2           ;FORM ENDING ADDRESS
        TST     -(R2)           ;LESS TWO
LOOP1:  MOV     PC,R4           ;SET UP RELATIVE ADDRESS
        ADD     #BUF-LOOP1-2+6,R4       ;OUTPUT BUFFER START ADDRESS
        MOV     #CR,R0          ;RESET PRINTER
        JSR     PC,PCHAR        ;CARRIAGE
        MOV     #LF,R0          ;FOR DUMP
        JSR     PC,PCHAR        ;INFORMATION
ARND:   MOV     R1,R0           ;CONVERT THE DUMP ADDRESS
        JSR     PC,CNVRT2       ;TO ASCII CHARACTERS
        MOV     #8.,R3          ;NUMBER OF WORDS DUMPED PER LINE
OCTAL:  MOV     PC,R4           ;SET UP RELATIVE ADDRESS OF
        ADD     #BUF-OCTAL-2+6,R4       ;BUFFER
        JSR     PC,CNVRT1       ;PRINT ONE WORD
        TST     (R1)+           ;NEXT ADDRESS TO BE DUMPED
        CMP     R2,R1           ;ARE WE DONE?
        BLO     FUDG            ;YES, PRINT A CR,LF
        SOB     R3,OCTAL        ;8 WORDS PRINTED? NO, GO BACK.
        BR      LOOP1           ;OTHERWISE START A NEW LINE
STOP:   HALT                    ;RESTING PLACE
;
;       THE PCHAR SUBROUTINE PUTS OUT ONE ASCII
;       CHARACTER WHICH IT FINDS IN R0
;
PCHAR:  TSTB    @#TPS           ;DEVICE BUSY?
        BPL     PCHAR           ;IF YES, KEEP TESTING
        MOVB    R0,@#TPB        ;PRINT A CHARACTER
        RTS     PC              ;AND RETURN
```

Рис. 7.2.

```
;
;       THE ACHAR SUBROUTINE ACCEPTS AN ASCII STRING AND CONVERTS IT
;       TO A 16-BIT WORD WHICH IS LEFT IN R5
;
ACHAR:  MOV     #5,CNT          ;MAX NUMBER OF ASCII DIGITS
        CLR     R5              ;OCTAL NUMBER IN INTERNAL FORM
AGAIN:  TSTB    @#TKS           ;DEVICE BUSY
        BPL     AGAIN           ;IF SO WAIT
        MOVB    @#TKB,R0        ;READ A CHARACTER
        JSR     PC,PCHAR        ;AND ECHO IT
        CMPB    R0,#123         ;TEST FOR AN "S" = 123 (8)
        BEQ     STOP            ;FIND IT--THEN HALT
        CMPB    R0,#215         ;FIND A CR
        BEQ     L1              ;IF FOUND THEN NUMBER IN R5
        ASH     #3,R5           ;SET UP R5 FOR NEXT DIGITS CONVERTED
        BIC     #177770,R0      ;ONLY NEED LOWER THREE BITS
        BIS     R0,R5           ;OR'ED INTO NUMBER FIELD
        DEC     CNT             ;DECREMENT CHARACTER COUNT
        BNE     AGAIN           ;GOT 5 CHARACTERS? NO-GO BACK
        MOV     #CR,R0          ;IF YES THEN
        JSR     PC,PCHAR        ;  SEND A CR
L1:     MOV     #LF,R0          ;    AND A LF
        JSR     PC,PCHAR
        RTS     PC              ;AND RETURN WITH VALUE IN R5
;
;       THE CNVRT SUBROUTINE CONVERTS A 16-BIT
;       WORD TO SIX OCTAL ASCII CHARACTERS
;
CNVRT1: MOV     (R1),R0         ;R0 CONTAINS OCTAL WORD TO BE CONVERTED
CNVRT2: MOV     #5,FIVE         ;ENTER HERE TO CONVERT THE CHARACTERS
L2:     MOVB    R0,R5           ;OCTAL BYTE ADDRESS OF LINE DUMPED
        BICB    #370,R5         ;STRIP OFF LOW ORDER THREE BITS
        ADD     #'0,R5          ;ADD IN ASCII ZERO
        MOVB    R5,-(R4)        ;TO OUTPUT BUFFER IN REVERSE ORDER
        ASH     #-3,R0          ;SHIFT OVER FOR NEXT THREE BITS
        DEC     FIVE            ;COUNT OCTAL CHARACTERS
        BNE     L2              ;HAVE WE DONE 5? NO, GO BACK
        MOVB    R0,R5           ;YES, SQUEEZE OUT LAST BIT
        BICB    #376,R5
        ADD     #'0,R5          ;ADD IN ASCII BASE
        MOVB    R5,-(R4)        ;STORE IT AWAY
DUMP:   TSTB    @#TPS           ;NOW DUMP THAT WORD
        BPL     DUMP            ;WAIT UNTIL FREE
        MOVB    (R4)+,@#TPB     ;MOVE IT TO TELEPRINTER
COMP:   MOV     PC,R0           ;CALCULATE LAST BYTE ADDRESS
        ADD     #BUF-COMP-2+7,R0;  OF BUFFER
        CMP     R0,R4           ;ARE WE DONE?
        BHIS    DUMP            ;NO, PRINT ANOTHER CHARACTER
        RTS     PC              ;YES, RETURN
FUDG:   MOVB    #CR,R0          ;PUT OUT THE CR
        JSR     PC,PCHAR
        MOVB    #LF,R0          ;AND LF COMBINATION
        JSR     PC,PCHAR
        JSR     PC,PCHAR
        JMP     ADDR            ;RETURN
;
;       CONSTANTS AND DATA BUFFERS
;
FIVE:   .WORD   0.
BUF:    .BLKB   6               ;SPACE FOR 6 ASCII CHARACTERS
        .ASCII  /               ;TWO SPACES WILL FOLLOW CHARACTERS
CNT:    .WORD   0
        .END CORE
```

Рис. 7.2. Программа восьмеричного дампа, использующая PIC

Выше приведена такая программа, обеспечивающая память на терминале восьмеричного дампа. На ее примере демонстрируются основы программирования вывода данных на терминал, а также использование позиционно-независимого кодирования. Применение PIC диктуется очевидной необходимостью иметь возможность загружать программу дампа в любое место памяти.

Программа начинает работу с вывода на печать (экран терминала) символа «A» и последующего ожидания ввода с клавиатуры восьмеричного числа (до 5 цифр), адресующего ячейку памяти, с которой начинается дамп. После нажатия клавиши ВК («возврат каретки») программа переводит строку и откликается выводом на экран символа «N». В ответ на этот символ надо ввести количество слов памяти, содержимое которых должно быть выведено на терминал. Теперь после нажатия ВК осуществляется вывод дампа. В примере ниже приведены первые 16 слов самой программы EDUMP:

```
A1000
N20

001000  010706  005746  005037  177560  005037  177564  112700  000015
001020  004767  000140  112700  000012  004767  000130  112700  000101
```

На рис. 7.1 показана ее блок-схема, а исходный текст — на рис. 7.2.

7.2.5. Высокоскоростное устройство ввода-вывода перфоленты

Высокоскоростная перфоленточная станция состоит из двух компонентов: устройства ввода и устройства вывода на восьмидорожечную перфоленту,— имеющих скорости 300 и 60 символов в секунду соответственно. Каждое из них имеет свои собственные регистры состояния и буферные регистры, позволяющие управлять вводом или выводом одного байта.

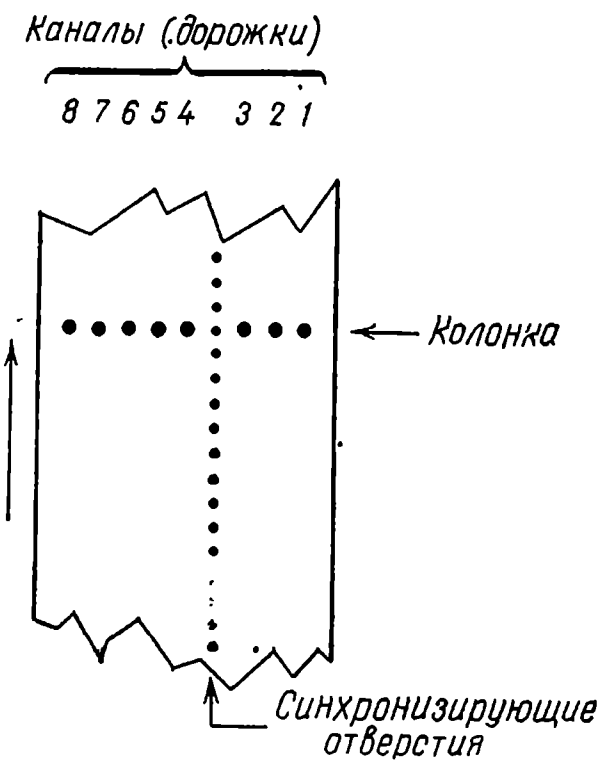


Рис. 7.3. Перфолента с пробивками

Данные записываются (набиваются) на перфоленте группами отверстий, расположенных в определенном формате по длине ленты. Лента, как это показано на рис. 7.3, делится на каналы, идущие вдоль нее, и на поперечные колонки.

Регистр состояния устройства чтения с перфоленты (см. ниже) по формату почти идентичен рассмотренному регистру состояния телетайпа TKS (ввод с клавиатуры/чтение). Отличие состоит лишь в бите ошибки, который в данном случае устанавливается при отсутствии ленты или если устройство находится в автономном режиме.

Функционирование устройства вывода на перфоленту во многом совпадает с работой телетайпа в режиме вывода на перфоленту/печать. Но перфоленточная станция имеет большую скорость вывода. Бит ошибки здесь такой же, как и у устройства чтения с перфоленты, т. е. устанавливается по сигналам «нет ленты» или «автономный режим».

7.3. ПРОБЛЕМА НАЧАЛЬНОЙ ЗАГРУЗКИ

При любом включении ЭВМ (кроме случаев восстановления после сбоя) ее память ничем не загружена. Если не рассматривать наличие аппаратного загрузчика (о нем будет сказано ниже), машина первоначально ничего «не знает», в том числе и как осуществлять ввод. Итак, проблема заключается в том, чтобы загрузить программу пользователя * в память.

Регистр состояния устройства чтения с перфоленты (PRS)

Бит	Назначение	Описание
15	Ошибка	Индицирует одно из трех возможных ошибочных состояний: нет ленты, устройство чтения в автономном режиме, нет питания. Запрещает работу (бит 0), вызывает прерывание, если бит 6 установлен
14—12		Не используются.
11	Занято (BUSY)	Индицирует, что символ читается. Очищается по команде INIT, и если символ в буфере, устанавливается командой RDR ENB.
10—8		Не используются.
7	Сделано (DONE)	Устанавливается, когда символ в буфере, очищается при ссылке на буфер (PRB) или по командам INIT или RDR ENA. Вызывает прерывание, если бит 6 = 1. Только чтение.
6	Прерывание разрешено	Разрешает прерывание, если бит DONE или бит ошибки установлен. Очищается по INIT.
5—1		Не используются.
0	Разрешение устройству чтения	Разрешает устройству чтения загрузить один символ. Очищает бит DONE; устанавливает «BUSY», очищает буфер (PRB). Очищается, если бит 15 = 1 (ошибка), и вызывает немедленное прерывание, если бит 6 = 1. Только запись.

* В первоначальный момент аналогично формулируется задача загрузки программ операционной системы. — *Примеч. ред.*

Регистр состояния устройства вывода на перфоленту (PPS)

Бит	Назначение	Описание
15	Ошибка	Индицирует одно из двух ошибочных состояний: нет ленты в устройстве вывода или на устройство не подано питание. Вызывает прерывание, если бит 6 = 1.
14—8		Не используются.
7	Готово (READY)	Устройство готово набить символ. Устанавливается по INIT, очищается при загрузке буфера и снова устанавливается, когда набит символ. Вызывает прерывание, если бит 6 = 1. Только чтение.
6	Прерывание разрешено	Вызывает прерывание, если установлены биты «READY» или «ошибка». Очищается по INIT.
5—0		Не используются.

Буфер устройства чтения с перфоленты (PRB)

Бит	Назначение	Описание
15—8		Не используются.
7—0	Данные	Хранит символ, который должен быть прочитан программой пользователя.

Буфер устройства вывода на перфоленту (PPB)

Бит	Назначение	Описание
15—8		Не используются.
7—0	Данные	Только запись. Любая команда, которая может модифицировать биты 7—0 буфера PPB, очищает «готово» и инициирует пробивку. Если же установлены биты 15 и 6 регистра PRS, то немедленно произойдет прерывание.

Очевидно, что для этого там уже должны находиться соответствующие команды, реализующие процедуры загрузки, т. е. мы сталкиваемся с некоторым парадоксом: для загрузки программы необходимо сначала загрузить другую программу. Именно поэтому рассматриваемая проблема получила название проблемы *начальной загрузки.*

Одно из возможных решений этой дилеммы — наличие в CPU некоторого механизма, который позволил бы пользователю *заносить* на хранение в специфицированные ячейки памяти машинные команды. Такой механизм должен включать возможность

спецификации как данных, которые заносятся в память, так и адресов, по которым эти данные записываются. Первые ЭВМ использовали переключатели и сигнальные лампочки в качестве «консольного» устройства I/O. На современных ЭВМ применяются клавишные панели и восьмерично/шестнадцатеричные устройства отображения. Некоторые большие мини-ЭВМ (например, VAX-11/780) используют в качестве консольного устройства терминалы на базе микропроцессоров.

Дисковые операционные системы, используемые в настоящее время, позволяют производить загрузку посредством короткой программы (загрузчика), которая читает нулевой блок с системного устройства в младшие адреса памяти, а затем выполняет эту только что прочитанную программу. Обычно такой загрузчик размещается в памяти только для чтения (ROM). Однако нам будет поучительно провести анализ перфоленточной системы загрузки. Это вызовет интерес не только у тех, для кого машина своего рода «хобби», но будет полезным и для получения навыков загрузки диагностических процедур в случае, если системное устройство дает сбой.

7.3.1. Программные загрузчики

Программный загрузчик (bootstrap) для PDP-11 — это последовательность команд для загрузки программы пользователя. Загрузчик использует специальный формат перфоленты и может

```
        077400          LOAD=77400      ;DATA MAY BE LOADED NO LOWER THAN THIS
        077744          .=77744         ;START ADDRESS OF THE BOOTSTRAP LOADER
077744  016701  START:  MOV     DEVICE,R1       ;PICK UP DEVICE ADDRESS
        000026
077750  012702  LOOP:   MOV     #LOOP-LOAD+2,R2 ;PICK UP ADDRESS DISPLACEMENT
        000352
077754  005211  ENABLE: INC     @R1             ;ENABLE THE PAPER TAPE READER
077756  105711  WAIT:   TSTB    @R1             ;WAIT UNTIL FRAME
077760  100376          BPL     WAIT            ;IS AVAILABLE
077762  116162          MOVB    2(R1),LOAD(R2)  ;STORE FRAME READ FROM TAPE
        000002
        077400
077770  005267          INC     LOOP+2          ;INCREMENT LOAD ADDRESS
                                                ;DISPLACEMENT
        177756
077774  000765  BRNCH:  BR      LOOP            ;GO BACK AND READ MORE DATA
077776  000000  DEVICE: 0                       ;ADDR OF INPUT DEVICE
                                                ;PUT HERE BY OPERATOR USING
                                                ;CONSOLE DEPOSIT SWITCH
        000001          .END
```

Рис. 7.4. Программный начальный загрузчик

самомодифицироваться. Исходный вид программы-загрузчика представлен на рис. 7.4. Заданный здесь начальный адрес говорит о том, что загрузчик должен загружаться в старшие адреса памяти (система с 16K). Вводится он вручную, с помощью переключателя в последние 14 слов памяти компьютера.

При работе загрузчик размещает прочитанные данные в последовательные байты, расположенные за адресом символа

LOAD. Образец перфоленты, загружающей данные с адреса 77600 по 77742, выглядит следующим образом:

```
351   —
351   |
351   |
 .    |
 .    | ← Лидер
 .    |
351   —
177   | ← Младший байт начального смещения минус 1
 .    —
 .    | ← Данные, которые должны быть загружены
 .    —
301   —
035   | ← Код, эквивалентный MOV DEVICE, R1
026   |
000   —
302   —
025   | ← Код, эквивалентный MOV #LOOP—LOAD + 2, R2
373   —
XXX   | ← Действительный адрес смещения между 77600 и 77742,
          по которому загруженная программа должна начать
          выполнение
```

Наличие специального лидера объясняется необходимостью загрузки нулей или пустой ленты. Загрузчик начинает свою работу с того, что загружает адрес регистра состояния устройства в R1 и 352(8) — в R2. Команда INC @R1 обеспечивает операцию чтения с устройства, следующие две команды организуют цикл ожидания до тех пор, пока эта операция не закончится. Когда байт данных принят, он пересылается в ячейку памяти, адрес которой представляет собой сумму индексного слова (77400) и содержимого R2. Пересылку делает команда MOVB 2(R1), LOAD(R2).

Так как первоначально в регистре R2 находится число 352(8), первый байт данных будет помещен в ячейку 77752, затем произойдет увеличение на 1 второго слова команды, помеченной символом LOOP (INC LOOP + 2) и выполнится переход на эту метку. Увеличенные на единицу данные (353(8)) становятся непосредственным значением, которое загрузится в R2, при очередном выполнении команды MOV # LOOP — LOAD + 2, R2.

Код лидера плюс увеличение на единицу в сумме дает значение числа, находящегося в R2 в момент начала загрузки (т. е. 352); поэтому код лидера не оказывает влияния на программу-загрузчик. Каждый раз, когда читается код лидера, процессор выполняет один и тот же цикл, и поэтому программа не модифицируется. Первый код, отличный от кода лидера, заменит данные в R2 некоторой величиной, которая будет восприниматься как указатель начала загружаемой программы с адресом загрузки, равным LOAD + (R2) = 77400 + (177 + 1) = 77600. Значит, следующие байты будут загружаться уже *не* по адресу непосредственного

операнда, а в ячейки памяти 77600, 77601, 77602, ..., 77742. Таким образом вся программа байт за байтом будет введена в память.

Как мы только что видели, команда INC при размещении данных в последовательных байтах требует, чтобы значение кода лидера и смещения в сумме были на 1 меньше, чем желаемая величина в R2.

В конце концов после загрузки последнего байта вводимой программы (а он записывается непосредственно перед загрузчиком) происходит наложение данных с перфоленты на две команды загрузчика. Первая из них (команда MOV DEVICE, R1) остается при этом без изменений, в то время как вторая модифицируется. Модификация осуществляется так, чтобы поместить очередной байт, вводимый с перфоленты (это смещение для перехода) в младший байт команды перехода. Для этого по команде MOV в R2 будет занесено число 374 (т. е. введенный байт, 373, увеличенный на 1), следовательно, в результате выполнения команды MOVB адрес 77400 + 374 = 77774 будет совпадать с младшим байтом команды BR LOOP. Благодаря занесению в эту команду нового смещения XXX загрузчик может передать управление на начало загруженной программы или на другой адрес внутри нее.

Таким образом, загрузка с самомодификацией применяется не только для ввода кодов программы, но и для инициализации кода самого загрузчика, а также для обеспечения принудительного перехода по адресу 77XXX внутри загруженной программы.

Основное требование к загрузчику связано с минимизацией занимаемой им памяти. Для наиболее полного удовлетворения этого требования применяется еще один способ загрузки программ пользователя.

Как было отмечено ранее, альтернативный метод — это добавление к CPU аппаратного загрузчика, чтобы можно было исполнять программу начальной загрузки (IPL) с помощью оборудования. IPL активизируется нажатием кнопки «загрузка» на CPU, вызывая тем самым выполнение предопределенной последовательности инструкций. Программа IPL включает как команды для вводного устройства, так и ячейки памяти, в которые должна помещаться информация.

Форма механизма загрузки меняется от машины к машине. Один механизм читает записи данных в память и выполняет как команду первое (или последнее) слово прочитанной записи. Другой выполняет последовательности команд, находящихся в памяти ROM (как в PDP-11) или на изменяемой панели «мертвого старта» (dead-start). Вне зависимости от метода результат обычно одинаков — загрузка в память короткой последовательности кодов, называемой *абсолютным загрузчиком*.

Абсолютный загрузчик — это системная программа для чтения записей, содержащих команды машинного языка и привязанных к абсолютным адресам ячеек памяти. В отличие от bootstrap'а абсолютный загрузчик может читать большие объемы информации

в различные сегменты памяти. Он использует формат, при котором каждая запись содержит:

1) счетчик количества слов в записи;
2) адрес загрузки, по которому будут загружены первое и последующие слова записи;
3) загружаемые коды;
4) адрес перехода для абсолютного загрузчика.

Оба загрузчика, абсолютный и bootstrap, являются системными программами. Это программы, сами по себе не производящие полезного результата, но помогающие программисту в реализации его планов. Они пишутся системными программистами, задача которых — предоставлять различные услуги пользователям системы. Системные программы включают, например, MACRO-11 и LINK. Гл. 8 целиком посвящена обсуждению некоторых из имеющегося большого количества системных программ, доступных пользователю PDP-11.

7.4. УСТРОЙСТВА ПАМЯТИ НА ЛЕНТАХ И ДИСКАХ

К малой ЭВМ, такой, как PDP-11, можно присоединить значительное количество внешних устройств с большим объемом памяти каждого. Наиболее часто эти ЭВМ комплектуются двумя такими устройствами, а именно DECtape (объем памяти 147968 слов) и DECdisk (объем памяти 1228800 слов). Стоит упомянуть о том, что устройства дисковой памяти особенно быстро развиваются и в настоящее время их объемы находятся в диапазоне от 256 Кбайт (диск RX01) до 176 Мбайт (диск RP07). Так как внешние устройства памяти большого объема требуют более тщательного программирования и управления, будет полезно провести анализ их характеристик и функционирования.

Ниже подробно рассматриваются два устройства: DECtape и RK05. Они выбраны потому, что являются представителями других, более современных устройств, в частности TU58 и RP07, и программирование их отвечает нашей цели изучения таких устройств.

7.4.1. Магнитная лента DECtape

DECtape содержит 10 дорожек, расположенных в соответствии с форматом, показанным на рис. 7.5. Первые пять — синхронизирующая, маркерная и три дорожки данных. Оставшиеся идентичны первым пяти и служат для повышения надежности системы; для этого на них записывается та же информация. Запись и чтение данных по информационным каналам происходит после записи по синхронизирующим и маркерным каналам. Маркер дорожки читается во время операции чтения и записи данных для того, чтобы определить функцию, которая должна выполняться системой в каждом управляющем режиме, и определить начало и конец блоков

данных. Три бита (по одному на каждой дорожке — это одна битовая позиция) составляют линию, или символ. Слово состоит из шести битовых позиций, или линий. Следовательно, на ленту можно записывать 18-битовые слова данных. Однако обычно два бита игнорируются.

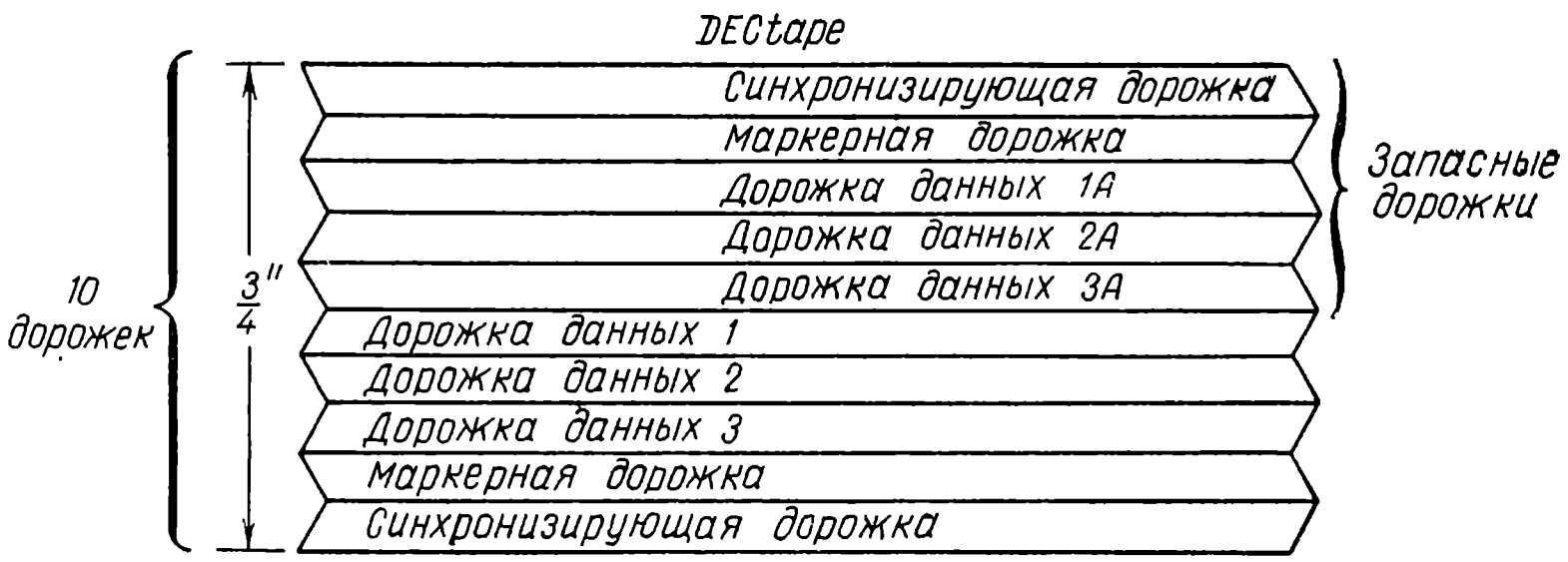


Рис. 7.5. Формат магнитной ленты DECtape

Кассета DECtape делится на три основные области: концевые зоны, зоны расширения и информационные зоны. Информационная зона состоит из блоков данных, содержащих по 256 слов. Как правило, лента содержит 578 информационных блоков (см. рис. 7.6).

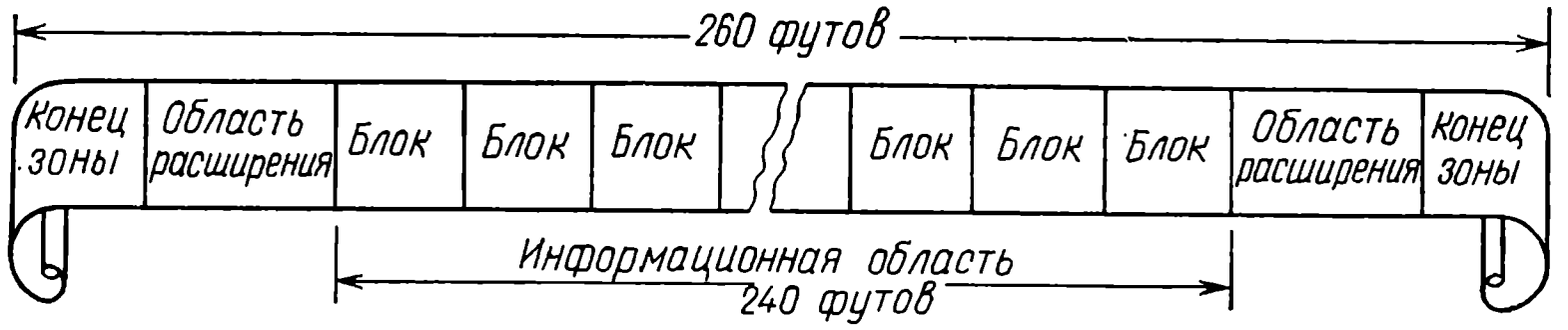


Рис. 7.6. Расположение блоков МЛ DECtape

Благодаря блочной структуре ленты цифровые данные на ней могут быть разделены на связанные группы слов, что уменьшает размер памяти, которая потребовалась бы при адресации каждого слова. Простым примером такой группы слов является программа. Она помещается на хранение и читается с магнитной ленты единым блоком, поскольку нет необходимости читать ее по одному слову. Каждая программа, таким образом, может быть помещена на ленту для хранения отдельным блоком.

Так как DECtape представляет собой систему с фиксированными адресами, то программисту нет нужды знать точно, где позиционирована лента. Чтобы найти нужное место, программист должен обеспечить только движение ленты в режиме поиска. Адрес текущего блока, который проходит под головкой чтения, считывается в контроллер и загружается в регистр интерфейса. Одновременно устанавливается флаг, который может вызывать программное прерывание. Программа сравнивает номер прочитанного блока

с желаемым адресом блока, и движение ленты продолжается или выполняется перемотка.

Работа ленты DECtape управляется контроллером посредством запрограммированных инструкций. Контроллер избирает лентопротяжку, управляет движением и направлением движения ленты, определяет операции чтения или записи, а также производит буферизацию передаваемых данных.

Он может выполнить любую из восьми команд, управляющих работой системы DECtape. В режиме on-line эти команды применяются для чтения и записи на ленту, а также для управления ее движением. Те из них, которые необходимы, задаются в программе. Чтобы занести восьмеричный код, представляющий команду, программист имеет возможность устанавливать или очищать биты 3, 2, 1 в регистре команд (TCCM).

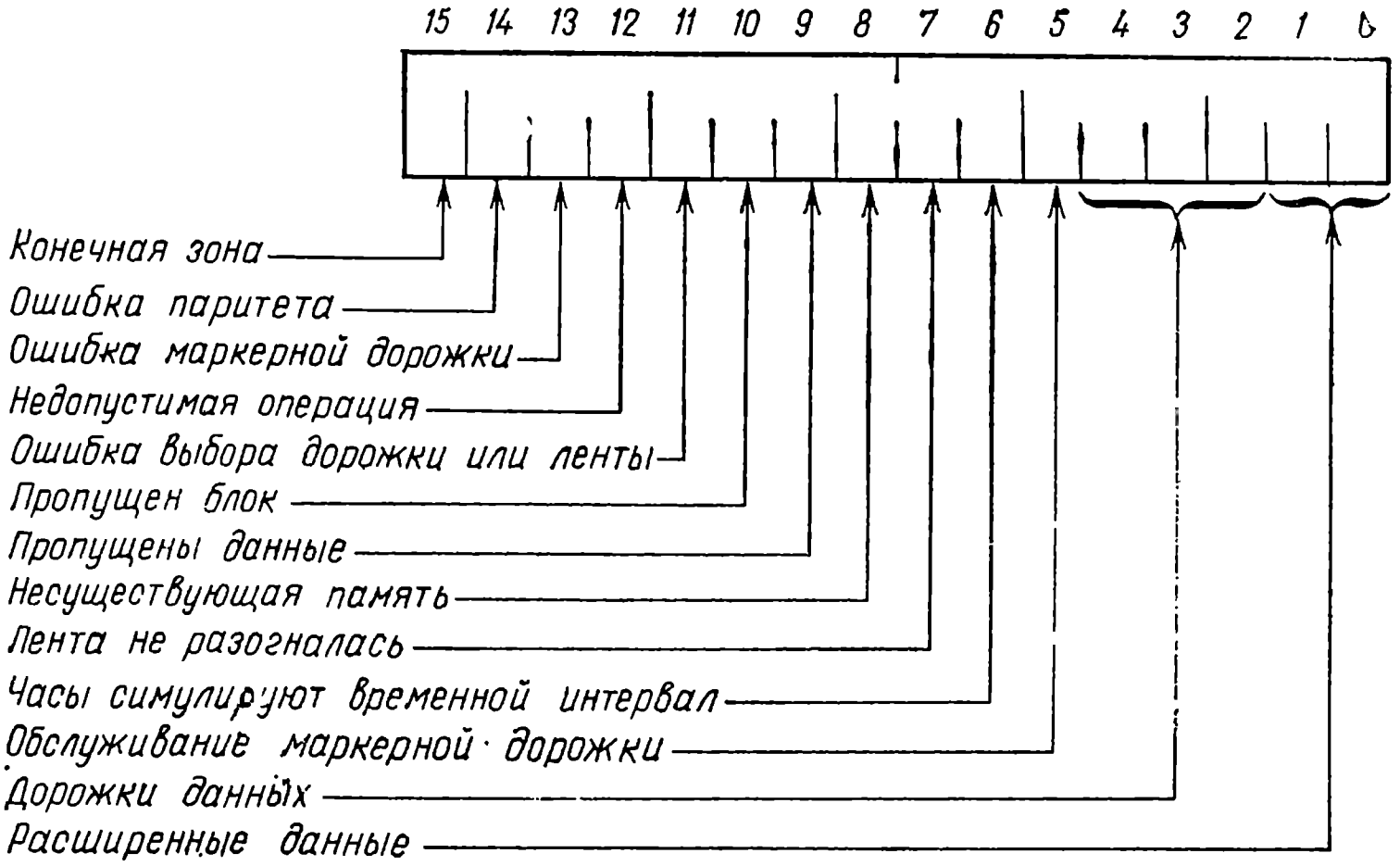
Код	Мнемоника	Функция
0	SAT	Остановить все лентопротяжки.
1	RNUM	Найти дорожку и номер нужного блока информации на ленте. Номер блока помещается в регистр данных (TCDT).
2	RDATA	Передает в память одно слово данных. Передача продолжается до переполнения счетчика слов, что означает конец текущего блока. Затем происходит проверка на четность.
3	RALL	Читается та информация с ленты, которая не может быть прочитана при выполнении функции RDATA.
4	SST	Остановить выбранную лентопротяжку.
5	WRTM	Записать информацию на синхронные и маркерные дорожки на чистой DECtape. Используется для форматирования новой ленты.
6	WDATA	Записать данные на три дорожки данных. 16 бит данных передаются прямо из памяти.
7	WALL	Записать информацию в те области ленты, которые недоступны функции WDATA.

Все программное управление системой DECtape осуществляется через пять регистров устройства. Их можно загружать или читать, используя любую команду PDP-11, которая ссылается на соответствующие адреса.

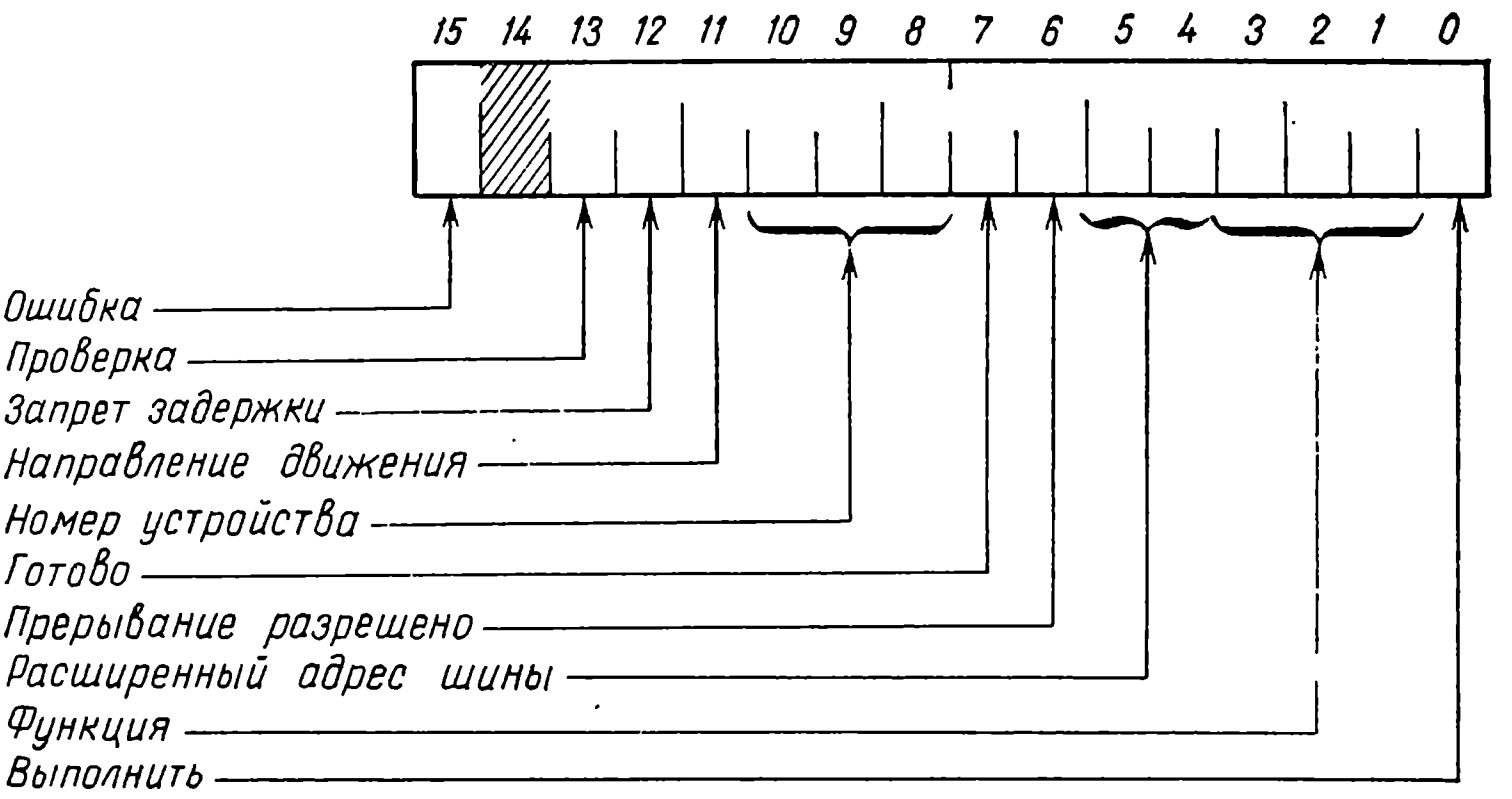
Регистр	Адрес
Регистр управления и состояния (TCST)	177340
Регистр команд (TCCM)	177342
Регистр-счетчик слов (TCWC)	177344
Регистр адреса шины (TCBA)	177346
Регистр данных (TCDT)	177350

Значения бит этих регистров представлены на рис. 7.7.

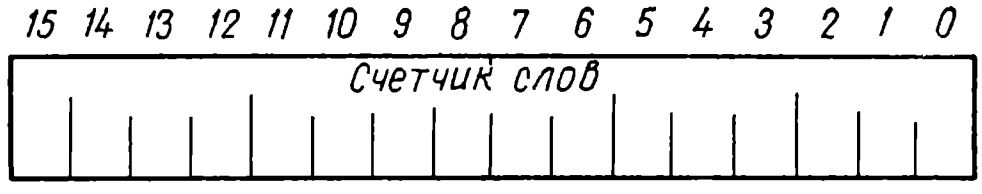
Регистр управления и состояния (TCST):



Регистр команд (TCCM):



Регистр-счетчик слов (TCWC):



Регистр адреса шины (TCBA):

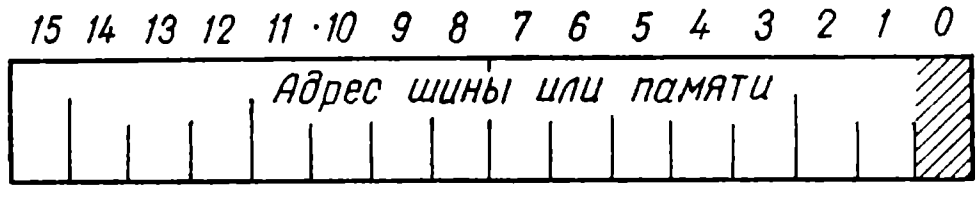


Рис. 7.7. Регистры управления и состояния МЛ DECtape

Регистры данных (TCDT):

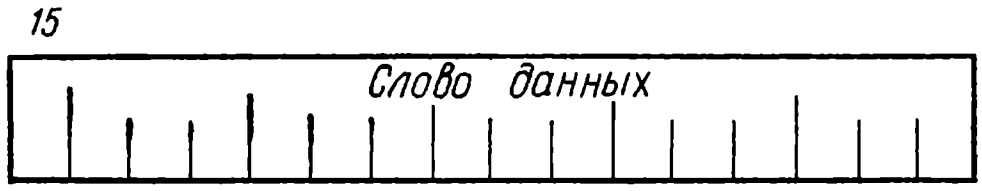


Рис. 7.7. Регистры управления и состояния МЛ DECtape (продолжение)

7.4.2. Примеры программирования операций ввода-вывода для DECtape

Поскольку ленты DECtape имеют организацию, подобную дискам, то и программируются они аналогично. Так, прежде чем записать блок, надо найти его место на ленте. Типичный метод на-

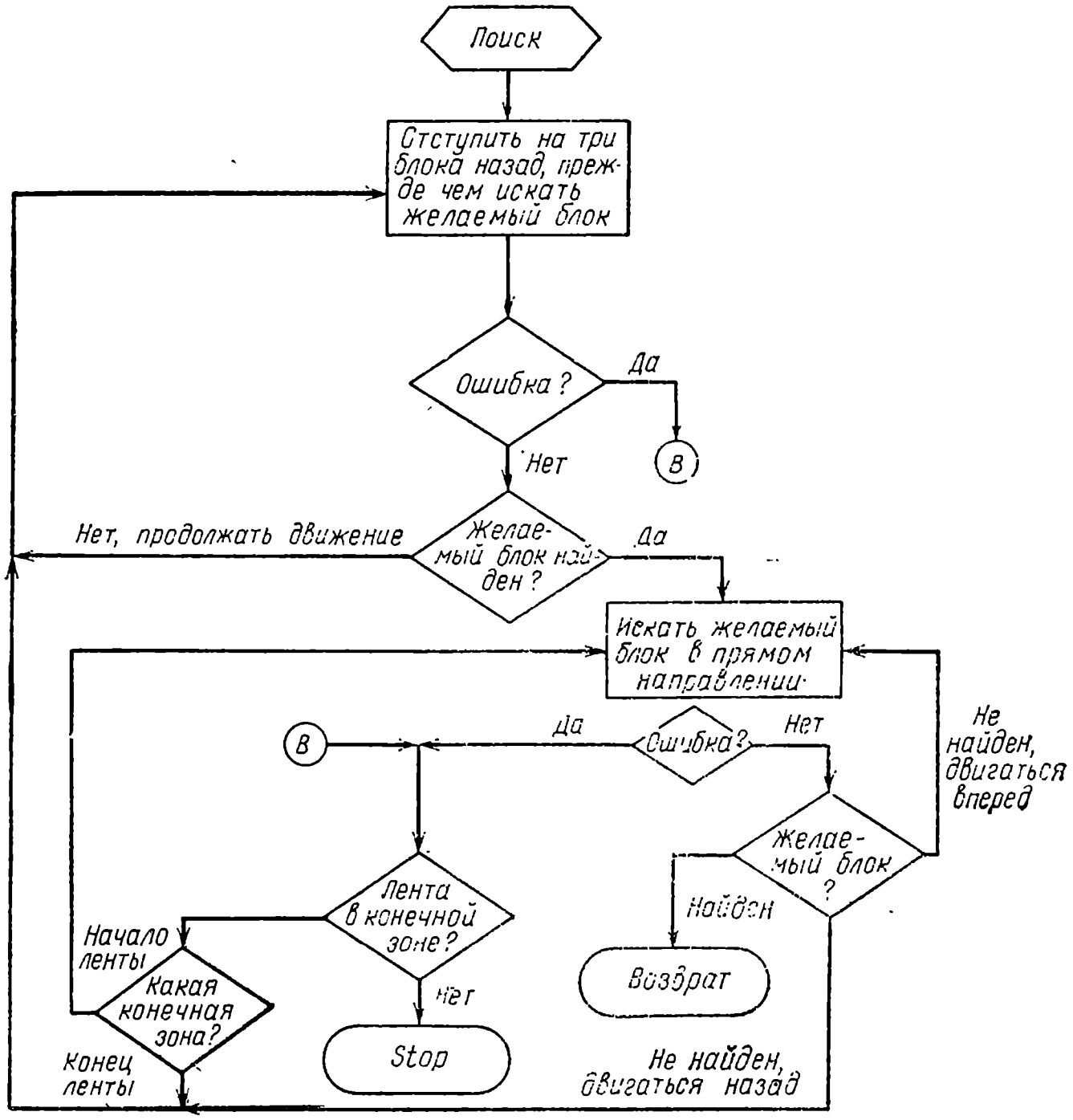


Рис. 7.8. Блок-схема для программирования МЛ DECtape

хождения блока — это инициация движения ленты и поиск желаемого блока при вращении ленты в прямом или обратном направлении. Операция заключается в сравнении номера каждого блока по мере того, как они читаются, с заданным. Как только блок найден, можно осуществлять запись или чтение его содержимого.

Эта процедура довольно проста, но тем не менее надо учитывать несколько особенностей DECtape. Первая из них заключается

в том, что, прежде чем записывать или читать, надо «разогнать» ленту. Это занимает некоторое время, и часть ленты пройдет под головкой чтения до того, как будет действительно прочитан номер

```
        ;DECTAPE SEARCH ROUTINE
        ;R0 CONTAINS DESIRED BLOCK NUMBER
        ;BLOCK FOUND IN FORWARD DIRECTION
;
        TCST=177340             ;CONTROL/STATUS REGISTER
        TCCM=TCST+2             ;COMMAND REGISTER
        TCDT=TCST+10            ;DATA REGISTER
;
SEARCH: MOV     R0,BWANT        ;SAVE BLOCK NUMBER
        SUB     #3,BWANT        ;OFFSET TO DESIRED BLOCK
        MOV     #4003,@#TCCM    ;READ BLOCK NUMBERS IN REVERSE DIR.
LOOP1:  BIT     #100200,@#TCCM  ;CHECK READY AND ERROR BITS
        BEQ     LOOP1           ;WAIT FOR READY
        BMI     ERROR           ;FOUND AN ERROR?
        SUB     @#TCDT,BWANT    ;CHECK BLOCK FOUND
        BLT     SEARCH          ;KEEP SEARCHING BACKWARDS
FORWRD: MOV     R0,BWANT        ;SAVE BLOCK NUMBER
        MOV     #3,@#TCCM       ;READ BLOCK NUMBERS IN FORWARD DIR.
LOOP2:  BIT     #100200,@#TCCM  ;CHECK READY AND ERROR BITS
        BEQ     LOOP2           ;WAIT FOR READY
        BMI     ERROR           ;HAVE AN ERROR?
        SUB     @#TCDT,BWANT    ;CHECK BLOCK FOUND
        BGT     FORWRD          ;BLOCK NUMBER TOO SMALL
        BLT     SEARCH          ;BLOCK NUMBER TOO BIG
        RTS     PC              ;RETURN WHEN BLOCK FOUND
ERROR:  TST     @#TCST          ;END ZONE ERROR?
        BMI     LOOP3           ;IF SO BRANCH
        HALT                    ;OTHERWISE HALT ON ERROR
LOOP3:  BIT     #4000,@#TCCM    ;TEST DIRECTION
        BNE     FORWRD          ;IF REVERSE, SEARCH FORWARD
        BR      SEARCH          ;IF FORWARD, SEARCH REVERSE
BWANT:  0                       ;BLOCK NUMBER
```

Рис. 7.9. Программа поиска для МЛ DECtape

первого блока. Вторая особенность состоит в том, что во время поиска номера блока под головку записи/чтения может подойти

```
        ;ROUTINE TO READ 100 WORDS FROM
        ;BLOCK 50, DECTAPE UNIT 0
        TCCM=177342             ;COMMAND REGISTER
        TCWC=TCCM+2             ;WORD COUNT REGISTER
        TCBA=TCCM+4             ;BUS ADDRESS REGISTER
;
START:  MOV     #START,SP       ;INITIALIZE STACK
        MOV     #50,R0          ;BLOCK 50 TO BE
        JSR     PC,SEARCH       ;  SEARCHED FOR
        MOV     #-100.,@#TCWC   ;COMPLEMENT OF WORD COUNT
        MOV     #BUFFER,@#TCBA  ;BUFFER ADDRESS
        MOV     #5,@#TCCM       ;READ DATA FORWARD DIRECTION
LOOP:   BIT     #100200,@#TCCM  ;CHECK ERROR AND READY
        BEQ     LOOP            ;WAIT FOR READY AND NO ERROR
        BMI     ERR             ;BRANCH ON ERROR
        .
        .
ERR:    HALT                    ;HALT ON ERROR
BUFFER: .BLKW 100.              ;SAVE ROOM FOR BUFFER
```

Рис. 7.10. Программа чтения для МЛ DECtape

зона конца ленты. Для выхода из этого положения необходимо включить реверс и осуществить поиск в обратном направлении. Третья, и последняя, особенность: после того как найдено нужное место (номер блока), непосредственно вслед за этим надо выпол-

нить запись или чтение, иначе передача данных закончится с ошибкой.

Учитывая сказанное, можно нарисовать блок-схему и составить код процедуры поиска, как это показано на рис. 7.8. Программа (1) выполняет проверку на допустимость требуемого номера блока, заданного в R0, и (2) осуществляет поиск этого блока в прямом направлении (рис. 7.9).

После того как специфицированный блок найден, осуществляется следующий шаг — передача информации. Программа на рис. 7.10 применяет подпрограмму SEARCH для поиска и читает 100 слов из блока 50 на устройстве с номером 0 ленты DECtape. Программа вызывает SEARCH, устанавливает счетчик слов, помещает адрес в буфер и затем ожидает, пока не закончится чтение. Читатель должен заметить, что хотя блоки и содержат по 256 слов (данных), но можно специфицировать любое их количество (до 256) для передачи.

7.4.3. Накопитель на магнитном диске

Обычно все дисковые устройства организованы на основе плоских магнитных поверхностей, называемых *тарелками* (platters). Поверхность делится на концентрические кольца, называемые *дорожками*, каждая из которых подразделяется на *секторы*. Сектор — наименьшая адресуемая единица, обычно он способен хранить значительное число машинных слов (например, 256-словный сектор). На рис. 7.11 показана организация такого диска.

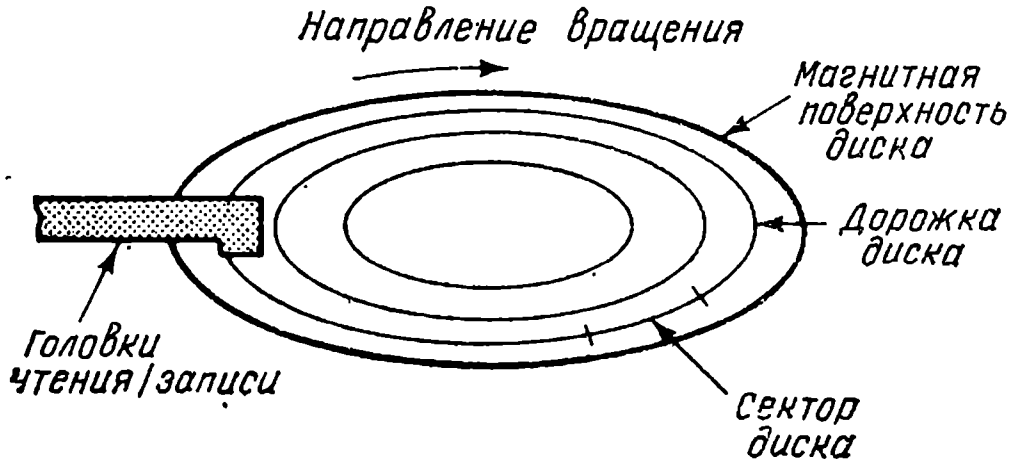


Рис. 7.11. Организация диска

В распоряжении пользователя малых машин имеется много альтернатив организации дисков. Это объясняется различными требованиями к таким устройствам. Диапазон выбора дисковых систем простирается от устройств с малым объемом памяти, но быстрым доступом до устройств с большим объемом памяти и средней скоростью доступа. Например, система дисков с *фиксированными головками* имеет головку чтения для каждой дорожки и, таким образом, удовлетворяет требованию к типу устройств с выгрузкой (swapping-type), т. е. таких, у которых содержимое памяти и диска должно быстро обмениваться. Эти устройства применяются в научных приложениях, где очень важны скорость доступа и передачи данных.

Если диск можно снимать с дисковода и заменять другим, то он называется *дисковым пакетом*. Пакет может состоять из одной или нескольких тарелок с множеством поверхностей, вертикально собранных на одной оси. Пакет дисков показан на рис. 7.12. Логически группируя все дорожки одинакового радиуса на каждой поверхности в *цилиндр*, мы можем увеличить доступный за одно обращение объем информации, повышая таким образом информационную плотность системы.

Системы с *подвижными головками* имеют единственную головку чтения/записи для всех дорожек одной поверхности. Поэтому время обмена с диском часто определяется продолжительностью механического поиска, связанного с физическим позиционированием

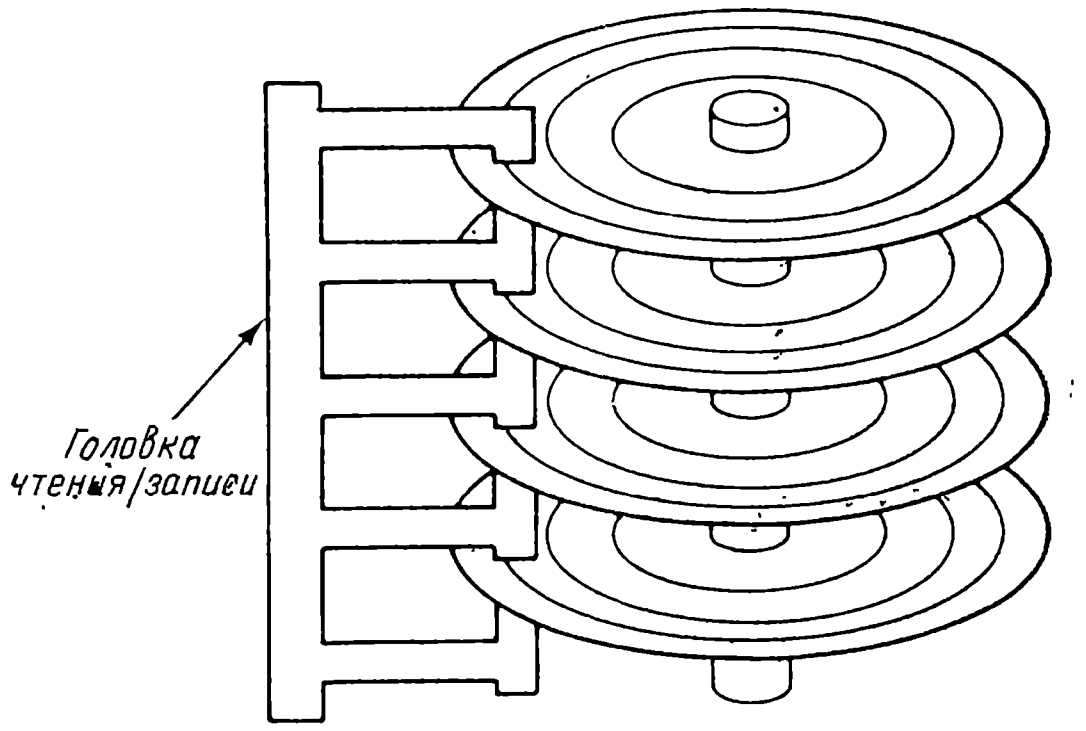


Рис. 7.12. Пакет дисков

блока головок на соответствующую дорожку. К этому следует добавить незначительное время на переключение электроники для выбора соответствующей головки. Таким образом, у системы с фиксированными головками меньше время доступа к данным, но она дороже стоит, ибо требует больше головок для чтения/записи (т. е. одну головку на дорожку). Такие системы идеальны для большой памяти. Потребность в эффективной с точки зрения стоимости системе подсказывает, что удобно иметь взаимозаменяемые дисковые пакеты, а это возможно только в системе с подвижными головками.

Мы считаем, что достаточно рассмотреть программирование только устройств, изображенных на рис. 7.11, так как управление пакетами дисков имеет много общего с управлением устройством с одной поверхностью.

Независимо от типа системы (с фиксированными или подвижными головками), есть еще одна деталь, замедляющая передачу данных. Она называется «*скрытым временем задержки*» (latency). Это время, необходимое для того, чтобы сектор прошел под головкой чтения/записи после ее выбора. Еще одно название такой задержки — «*задержка вращения*» (rotational delay). По смыслу она подобна задержке при использовании ленты, когда надо ожидать, пока нужный блок информации пройдет под головкой чтения/записи.

Время задержки может быть сокращено за счет увеличения скорости вращения дисков. При этом большее количество информации будет проходить под головкой чтения/записи в заданный промежуток времени. Таким образом увеличивается число передаваемых в секунду символов, или *темп передачи* устройства. Темп передачи может быть увеличен и за счет большего количества информации, помещаемого на дорожку (например, увеличение *плотности* записи информации). Все эти факторы, т. е. плотность, темп передачи, скрытая задержка, время поиска, фиксированная или подвижная головка, число поверхностей дисков и т. д., должны быть приняты во внимание при выборе оптимальной дисковой системы для решения конкретных проблем.

7.4.4. Программирование операций ввода-вывода для DECdisk

Для простоты мы будем рассматривать пакет дисков RK05 с подвижными головками, сектор которого содержит 256 слов, дорожка — 12 секторов, одна поверхность — 200 дорожек и общий

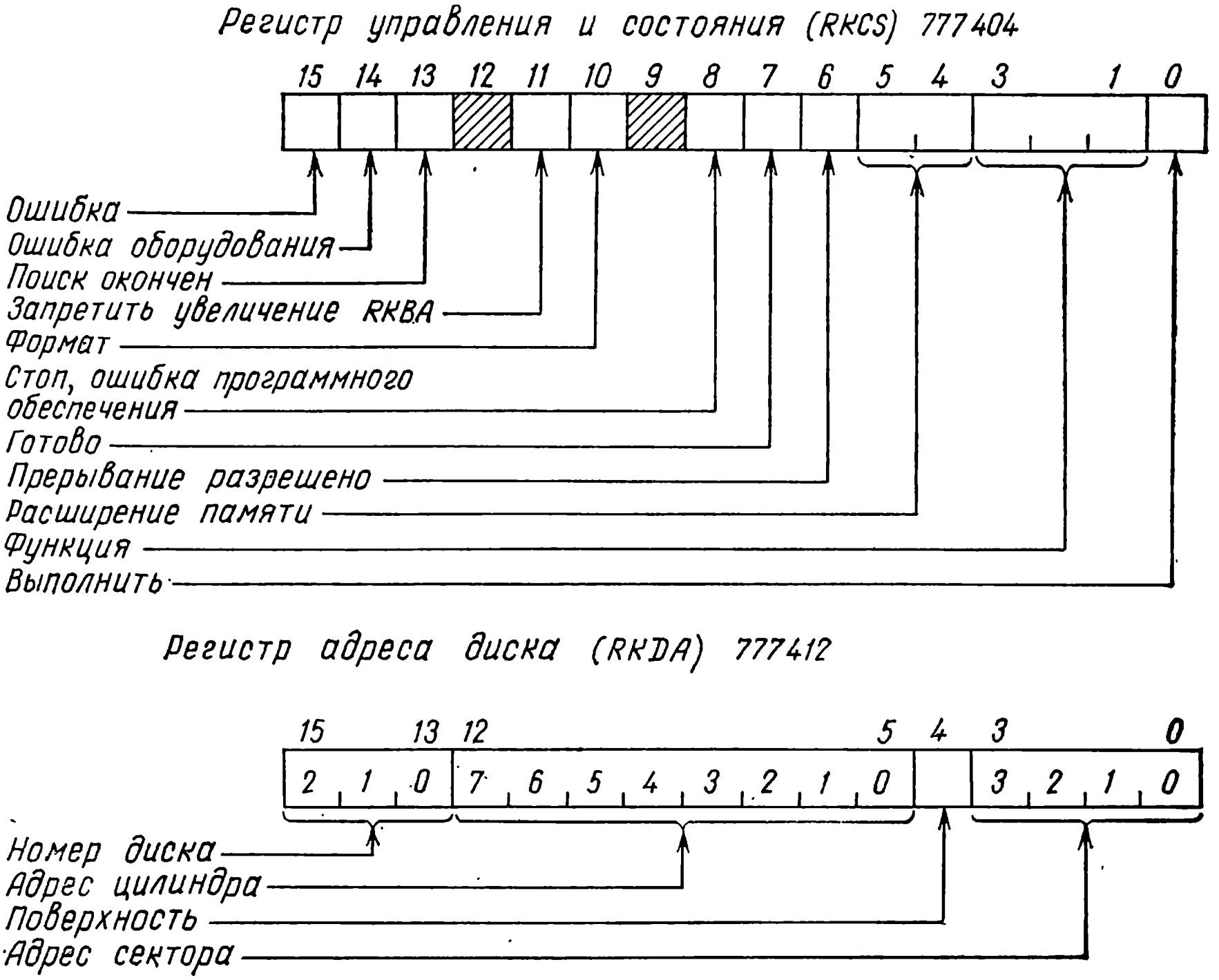


Рис. 7.13. Регистры RK05

объем памяти равен 1228800 слов. Управление этим пакетом осуществляется программно, с использованием 8 регистров. Подобно регистрам других устройств I/O, регистры RK05 можно читать или

писать в них, используя любые команды PDP-11, ссылающиеся на их адреса:

Регистр	Адрес
Регистр состояния диска (RKDS)	177400
Регистр ошибки (RKED)	177402
Регистр управления и состояния (RKCS)	177404
Регистр-счетчик слов (RKWC)	177406
Регистр текущего адреса шины (RKBA)	177410
Регистр адреса диска (RKDA)	177412
Регистр-буфер данных (RKDB)	177416

Назначение битов для подмножества этих регистров показано на рис. 7.13.

Количество слов, которое надо передать, записывается (с дополнением до двух) в RKWC; в RKBA загружается адрес памяти, начиная с которого будет производиться передача. Содержимое RKBA увеличивается после каждой передачи. Хотя с диском связано больше регистров, чем с лентой, тем не менее программирование их легче, так как здесь нам не нужно давать команды на старт, останов или реверс диска для поиска нужного сектора. Поскольку диск вращается с постоянной скоростью, все, что от нас требуется,— это установить адрес нужного сектора, счетчик слов, адрес некоторого буфера и затем ждать, пока не произойдет передача данных. Эта последовательность операций может быть запрограммирована так, как это показано на рис. 7.14.

```
;PROGRAM TO READ 512. WORDS FROM
;DISK UNIT 0, CYLINDER 4, SURFACE 1, SECTOR 7 TO MEMORY
        RKCS=177404                     ;DISK CONTROL/STATUS REGISTER
        RKWC=RKCS+2                     ;DISK WORD COUNT REGISTER
        RKBA=RKCS+4                     ;CURRENT BUS ADDRESS REGISTER
;       RKDA=RKCS+6                     ;DISK ADDRESS REGISTER
START:  MOV     #227,@#RKDA             ;DRIVE, CYLINDER, SURFACE, SECTOR
        MOV     #-512.,@#RKWC           ;WORD COUNT
        MOV     #BUFFER,@#RKBA          ;BUFFER ADDRESS
        MOV     #5,@#RKCS               ;INITIATE READ
BUSY:   TSTB    @#RKCS                  ;CONTROLLER BUSY?
        BPL     BUSY                    ;YES
        TST     @#RKCS                  ;ANY ERRORS?
        BMI     RKERR                   ;IF NEGATIVE, YES
        .
        .
RKERR:                                  ;CHECK ERRORS.
        .
        .
BUFFER: .BLKW 512.
```

Рис. 7.14. Программа чтения с диска

Как и при работе с лентой, в этом случае также можно передавать любое количество слов (насколько позволяет память). Регистр дискового адреса автоматически увеличивается после каждой передачи сектора. Этот процесс идет по всем цилиндрам, поверхностям и даже дисковым устройствам. Если же надо передать часть сектора (меньше, чем 256 слов), то следует установить в регистре-счетчике слов соответствующее значение и только затребованное количество слов будет участвовать в обмене.

Полезно проанализировать программу начальной загрузки с RK05, выполняющую чтение первого сектора системного диска в память.

```
012700          MOV     #177406,R0      ;R0 ← ADDRESS OF WORD COUNT REG.
177406
012710          MOV     #177400,(R0)    ;MOVE -256. TO WORD COUNT REGISTER
177400
012740          MOV     #5,-(R0)        ;MOVE 5. TO CSR REGISTER. (READ!)
000005
105710  T:      TSTB    (R0)            ;TEST FOR READ COMPLETION
100376          BPL     T               ;WAIT FOR IT
005007          CLR     R7              ;GO TO LOCATION 0 TO EXECUTE
                                        ;   256 WORD PROGRAM JUST READ IN
```

Прежде чем начнет работать программа-загрузчик, необходимо с консоли выполнить сброс системы (RESET). Эта операция обнуляет все периферийные регистры, включая RKBA и RKDA, и, следовательно, данные читаются в главную память, начиная с ячейки 0, а дисковая память типа RK05 адресуется устройством 0, цилиндром 0, поверхностью 0, сектором 0. При таком подходе размер программы-загрузчика существенно уменьшается. Последняя инструкция (CLR R7) передает управление на ячейку памяти 0, т. е. на начальный адрес 256-словной программы, только что прочитанной с диска.

7.5. ПРЕРЫВАНИЯ

При выполнении на машине задачи, использующей программы ввода и вывода, время тратится в основном на ожидание, пока информация будет передана на устройство I/O или прочитана с него. Иными словами, оно тратится на тестирование или опрос регистра состояния устройства и последующего ожидания в цикле выполнения условия «сделано» (DONE).

```
TEST:   TSTB    @#TKS           ;TEST CSR
        BPL     TEST            ;WAIT FOR DONE
```

В таком цикле расходуется масса машинного времени. Если исключить цикл ожидания, то за счет этого ЭВМ могла бы выполнять другие работы. Для реализации такой возможности и одновременно совместного использования машины несколькими программами или задачами служит механизм прерываний, стандартный для всех серий ЭВМ PDP-11.

Смысл прерывания состоит в предоставлении возможности процессору автоматически реагировать на внешние и внутренние воздействия, обусловленные различными причинами. Непредсказуемые во времени события (такие, как окончание операции I/O) могут генерировать прерывания и принуждать машину выполнять *программу обработки прерываний* (ISR). Таким образом, пользователю нет необходимости тестировать или опрашивать регистры после выполнения каждой команды, чтобы узнать, произошло ли

то или другое событие. Он может написать ISR, которая будет выполняться именно тогда, когда оно произойдет.

Практически прерывание — это скачок на подпрограмму, «насильно» вызванный оборудованием, в противоположность переходу, который точно специфицируется как переход к подпрограмме на основе выполнения команды перехода. Прерывания могут возникнуть в любой точке программы (перед выполнением I-фазы очередной команды) и должны информировать систему о соответствующей причине. Например, когда в некоторых машинах случается прерывание, то в специальном регистре прерываний устанавливается специальный бит, указывающий на вызвавшее его условие. В ответ на это CPU выбирает адрес следующей команды из конкретной (фиксированной) ячейки прерываний (возможно, имеющей адрес памяти нуль) и начинает выполнение программы анализа прерывания.

7.5.1. Прерывания и работа программ

Подобно программам, прерывания имеют информационные связи для возврата управления прерванной программе. Однако из-за того, что природа прерываний случайна, объем запоминаемой информации при прерываниях значительно больше, чем при обычных переходах из одной подпрограммы в другую. Здесь необходимо полное сохранение машинного состояния прерванной программы, иначе могут появиться нежелательные искажения ее параметров состояния (т. е. кодов условий, содержимого PC и SP и т. д.). При правильной обработке прерывание не вызывает потери какой-либо информации (на это только тратится время) и оно остается «невидимым» для прерываемой программы. Временная задержка такой программы равна интервалу между ее остановом, работой ISR и последующим возобновлением выполнения *.

7.5.2. Состояние машины при прерывании

Полное машинное состояние программы, непосредственно выполнявшейся перед прерыванием, обычно запоминается в *слове состояния процессора* (PSW). В машине с большой длиной слова памяти в PSW наряду с кодами условий сохраняется и счетчик команд. На миникомпьютерах же, таких, как PDP-11, приходится PSW разделять на две части, чтобы разместить информацию о *состоянии процессора* (PS) и содержимое счетчика команд (PC).

Техника манипуляции прерыванием состоит в том, что текущее PSW заменяется на PSW программы, обрабатывающей прерывание. А текущее PSW сохраняется в некотором слове памяти.

---

* Разумеется, при условии, что после обработки прерывания будет запущена эта же, а не другая программа. — *Примеч. ред.*

Диаграмма этого процесса представлена на рис. 7.15. Из рисунка видно, что для процесса прерывания требуются две ячейки памяти плюс регистр, в котором будет находиться текущее PSW.

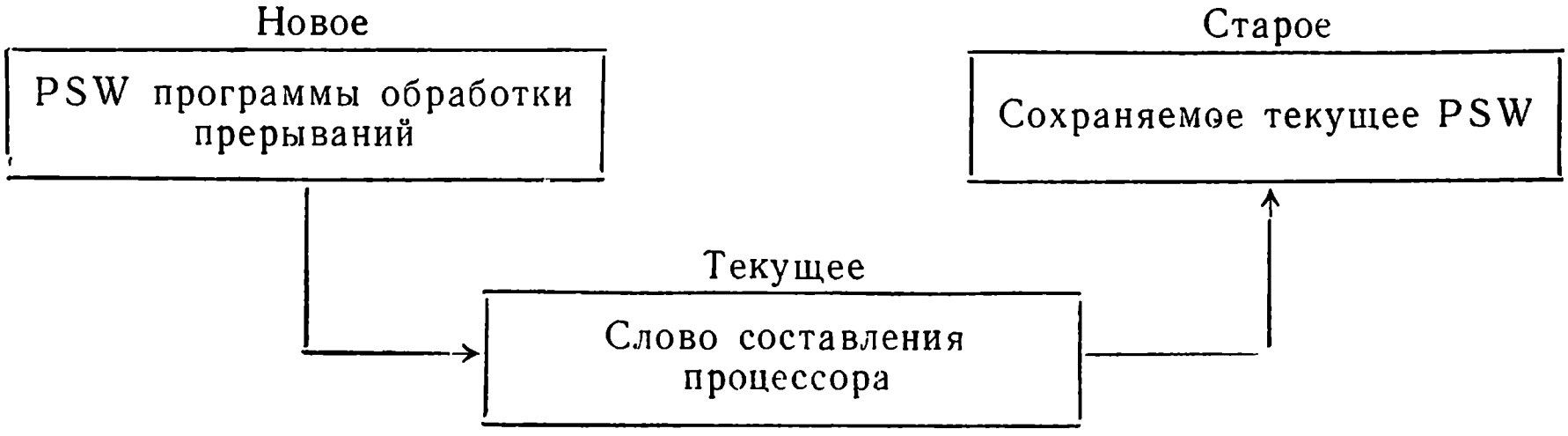


Рис. 7.15. Изменение состояния процессора

7.5.3. Сохранение прерываний

При использовании приведенной схемы возникает проблема, заключающаяся в том, что все прерывания применяют одни и те же ячейки для запоминания и восстановления. Поэтому, случись второе прерывание во время выполнения программы обработки первого, будет потеряно старое PSW, так как на его место будет записано PSW очередной прерванной программы. Чтобы этого не происходило, надо запретить новые прерывания, пока не будут обработаны предыдущие. Это можно сделать просто, отведя для этой цели один бит разрешения/запрещения прерываний в PSW, устанавливая его в 1 в старом PSW и сбрасывая в новом. Тогда, как только новое PSW становится текущим, прерывания запрещаются, а при *возврате из прерывания* (RTI) текущее PSW заменяется старым (сохраняемым), и они снова разрешаются.

В период времени, когда прерывания запрещены, вполне возможно, что в системе возникнут новые условия, например по I/O, которые вызовут прерывание. Им не дается разрешения на немедленный вызов программы обработки, они запоминаются и хранятся в *регистре состояния прерываний*. Следовательно, когда прерывания запрещены, они могут происходить, и есть гарантия, что впоследствии они будут обработаны.

Регистр состояния прерываний служит многим целям. Во-первых, он индицирует причину прерывания. Во-вторых, сохраняет вновь возникшие условия прерываний на период времени, пока их обработка запрещена. И в-третьих, обеспечивает гибкость обработки прерываний, ибо программист сам решает, какое из них нужно обрабатывать следующим. Это особенно удобно, если поддерживается относительный приоритет различных прерываний. В этом случае, если одновременно произошло несколько прерываний, программист может потребовать обработку первым самого критического.

Однако передача программисту прав назначения приоритета может привести к другим проблемам. Так, если высокоприоритетное прерывание произошло в момент, когда они запрещены, то нет

способа обработать его до тех пор, пока не будет снова выполняться программа анализа прерываний. Таким образом, приходится снова задействовать весь механизм их обработки во время работы с текущим прерыванием. Следовательно, чтобы схема все же работала, надо помещать в стек информацию о прежнем прерывании (старое PSW) и устанавливать бит разрешения прерываний в новом PSW. Однако и в этом случае остается проблема, связанная с устройствами, вызывающими прерывания.

7.5.4. Приоритетность прерываний

Раз бит разрешения прерываний установлен, любое устройство может вызвать прерывание. Поэтому программа анализа прерываний должна производить регистрацию их состояния, чтобы иметь возможность выбора для обработки прерывания от самого

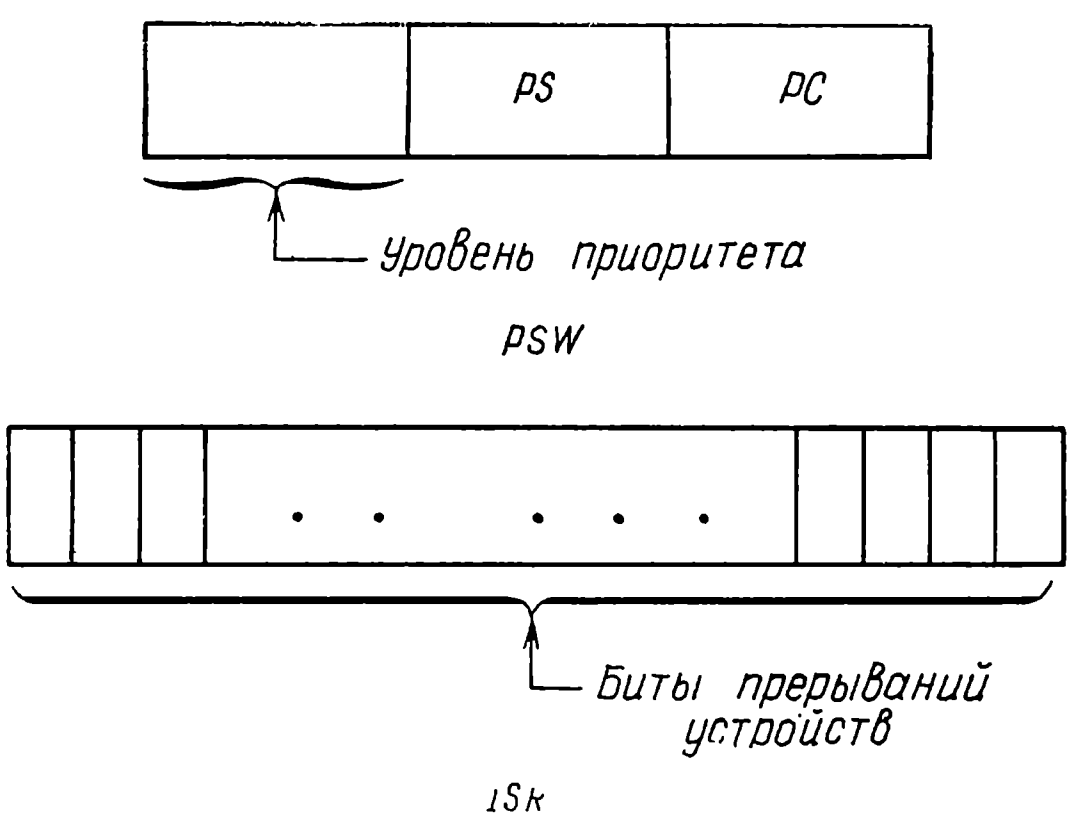


Рис. 7.16. Слово состояния процессора и слово состояния прерываний

высокоприоритетного устройства. Для этого можно использовать следующую процедуру *: если в какой-либо момент времени обрабатывается прерывание некоторого уровня приоритета, то его обработка может быть прервана только при возникновении более приоритетного прерывания, тогда как аналогичные им менее приоритетные сохраняются для последующей обработки. В этом случае для удобства программиста приоритетность устройств и программ обработки может быть встроена в оборудование и при такой схеме *приоритетности прерываний* целую группу устройств можно считать выполняющейся на определенном приоритетном уровне.

На рис. 7.16 показаны типичные PSW и регистр состояния прерываний. На их основе можно построить довольно изощренную схему прерываний, однако в любой из них на программиста по-

* Такая дисциплина обслуживания обычно называется дисциплиной с абсолютными приоритетами. — *Примеч. ред.*

прежнему возлагается обязанность определять причину прерывания. Схема будет лучше, если каждой группе устройств одинакового уровня приоритетности назначить PSW своей программы обработки прерываний из постоянной ячейки, закрепленной за этой группой. В этом случае, если произойдет прерывание, будет априори известна группа устройств, вызвавшая его.

7.5.5. Схема автоматической установки приоритетности прерываний

В развитие этой идеи хорошо бы каждому устройству иметь свое PSW. Тогда на 100 устройств нужно иметь 100 новых PSW (и 100 старых), указывающих на 100 потенциально различных программ обработки прерываний (ISR). И так как каждое прерывание инливидуально идентифицируется, нет необходимости иметь регистр состояния прерываний, а значит, и программу анализа прерываний. В результате мы сохраняем время и память, которые, однако, будут целиком исчерпаны большим количеством PSW, которые надо разместить в памяти.

В PDP-11 мы можем увидеть модифицированную версию этой схемы *автоматического задания приоритетности прерываний.* Для запоминания информации о состоянии машины ЭВМ используют два слова: состояние процессора и счетчик команд. При прерывании текущее содержимое PS и PC автоматически «проталкиваются» в системный стек, реализуемый с помощью SP (6-й регистр). Эффект тот же, что и при выполнении команд.

```
MOV     PS,-(SP)        ;PUSH PROCESSOR STATUS
MOV     PC,-(SP)        ;AND PROGRAM COUNTER
```

Новое содержимое PC и PS загружается из двух определенных последовательных ячеек памяти, которые вместе называются *вектором прерываний.* Эти ячейки выбраны дизайнером интерфейса устройств и располагаются в младших адресах памяти. Первое слово содержит адрес программы обработки прерываний (адрес новой программной последовательности), а второе — новое PS, на основе которого определяются состояние машины и уровень приоритетности. Содержание этих векторов определяется программистом и может меняться под управлением программы.

После того как ISR заканчивает выполнение командой RTI (возврат из прерывания), из стека автоматически «выталкиваются» два верхних слова и помещаются в PC и PS соответственно, возобновляя таким образом выполнение прерванной программы. Так как стек используется механизмом прерываний автоматически, прерывания могут «гнездоваться» подобно тому, как это происходит при передаче управления во вложенных подпрограммах. Естественно прерывания могут чередоваться с переходами к подпрограммам, причем никаких недоразумений при этом не происходит. Использование команд RTI и RTS обеспечивает автоматический возврат в соответствующие программы.

7.5.6. Обработка прерываний от устройства ввода перфоленты

В качестве примера выполнения действия по прерываниям рассмотрим операции, происходящие на PDP-11 при работе программы чтения в буфер блока символов с устройства ввода перфолен-

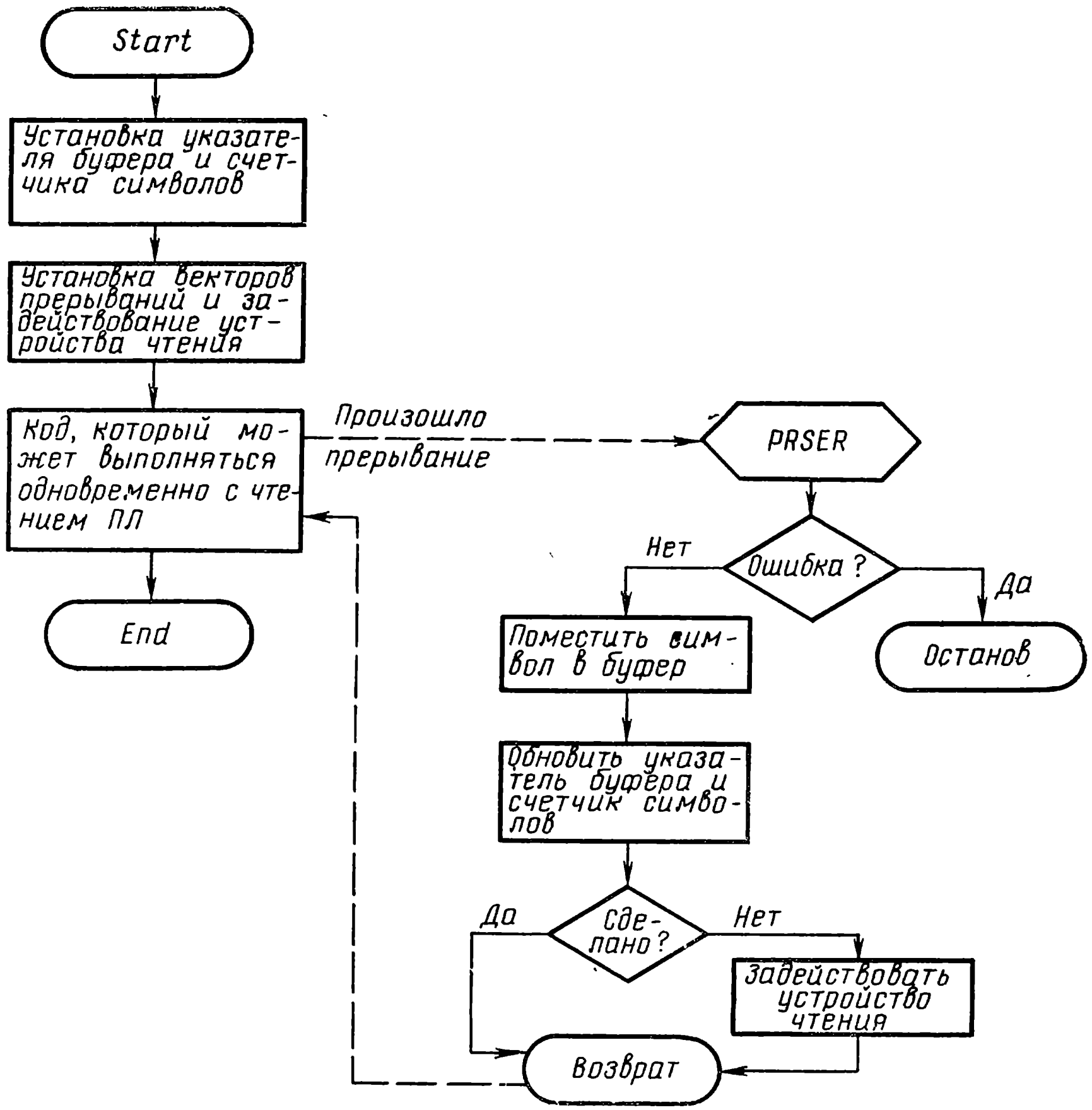


Рис. 7.17. Блок-схема процесса прерываний при чтении с перфоленты

ты. Соответствующая блок-схема показана на рис. 7.17. Программа написана в коде PIC и включает установку вектора прерываний (ячейка памяти 70) для устройства ввода перфоленты. В ней используются две отдельные процедуры. Первая, начинающаяся с метки INIT, инициализирует счетчик слов для процедуры обработки прерываний и затем вычисляет фактор перемещаемости на основе смещения PRSER — X — 2 следующим образом.

Пусть PCz есть PC, которое было бы установлено, если бы программа загружалась с нулевого адреса, а PCn — текущее значение счетчика команд. Тогда

```
PRSER - X - 2 + PCn = PRSER - PCz + PCn
                    = PRSER + (PCn-PCz)
```

Здесь PCz = (X + 2). В результате фактор перемещаемости PCn — —PCz добавляется к значению PRSER, определенному при ассемблировании, и получается перемещаемое значение PRSER. Перемещаемое значение BUFADR вычисляется подобным образом.

Затем программа устанавливает уровень приоритета для устройства чтения и устанавливает разрешение прерывания после того, как символ прочитан. Такая логика работы освобождает CPU, и он может выполнять какие-то другие программы, пока заполняется буфер.

Вторая процедура — это программа PRSER. Она обрабатывает прерывания от устройства чтения перфоленты. PRSER активизируется каждый раз, как получен символ, посылает символ в буфер, обновляет указатели буфера и счетчика слов и снова восстанавливает готовность устройства, если продолжается чтение. При выполнении операции может оказаться, что количество символов меньше, чем 100, или возникает ошибка чтения с перфоленты. Оба эти условия вызывают состояние ошибки. При этом необходимо выполнить возврат из процедуры с кодом «функция выполнена» и указание причины ошибки. На рис. 7.18 CRCNT передает — 1 в вызывающую программу, сигнализируя таким образом о возникновении ошибки. Логика этих двух программ показана на рис. 7.17, а их коды — на рис. 7.18.

7.5.7. Уровни приоритетов и маскирование прерываний

На конкретном уровне приоритетности каждое из устройств внутри группы способно вызвать прерывание. Можно допустить, что программист на какой-то период времени пожелает игнорировать прерывания от некоторых из них. Поэтому оборудование обычно включает механизм селективного маскирования прерываний.

На PDP-11 применяется простой механизм маскирования. Программист, очищая бит «прерывание разрешено» в регистре управления и состояния устройства, запрещает, таким образом, прерывание. В действительности бит прерывания автоматически очищается каждый раз, когда система инициализируется (нажатием ключа START или выполнением команды RESET), и должен устанавливаться под управлением программы. Однако, будучи однажды установлен, бит остается в этом состоянии до новой очистки.

Маскировать прерывания можно еще одним способом — применять *регистр маски.* Такой регистр имеет бит для каждой группы

устройств (что эквивалентно биту регистра состояния прерываний). Оборудование же, используя биты маски, выполняет логическую операцию AND с битами прерываний. И только если результат равен 1, то возможно прерывание.

```
;INTERRUPT DRIVEN ROUTINE TO INPUT CHARS FROM PAPER TAPE READER
        PRS=177550                      ;DEFINE DEVICE
        PRB=PRS+2                       ;REGISTERS
;
INIT:   MOV     PC,SP                   ;INITIALIZE STACK
        TST     -(SP)                   ;TO POINT TO INIT
        MOV     #100.,CRCNT             ;SET UP CHARACTER COUNT
;
X:      MOV     PC,R0                   ;R0 ← ADDR(X+2)
        ADD     #PRSER-X-2,R0           ;ADD OFFSET TO GET ISR ADDRESS
;
Y:      MOV     PC,PTR
        ADD     #BUFADR-Y-2,PTR         ;SET UP BUFFER ADDRESS POINTER
;
        MOV     R0,@#70                 ;SET UP ISR VECTOR ADDRESS
        MOV     #200,@#72               ; STATUS TO PRIORITY 4
;
        MOV     #101,@#PRS              ;SET INTR ENB AND RDR ENB
        .
        .               CODE WHICH MAY BE EXECUTED WHILE
        .               BUFFER IS BEING FILLED
        .
BUFADR: .BLKB 100.                      ;100 CHARACTER BUFFER
PRSER:  TST     @#PRS                   ;TEST FOR ERROR
        BMI     ERROR                   ;DO ERROR THING
        MOVB    @#PRB,@PTR              ;STORE CHARACTER IN BUFFER
        INC     PTR                     ;BUMP POINTER
        DEC     CRCNT                   ;DECREMENT CHARACTER COUNT
        BEQ     DONE                    ;BRANCH WHEN INPUT DONE
        INC     @#PRS                   ;START UP READER AGAIN
DONE:   RTI                             ;RETURN
ERROR:  MOV     #-1,CRCNT               ;RETURN ERROR CODE
        RTI                             ;RETURN
PTR:    0                               ;BUFFER POINTER
CRCNT:  0                               ;CHARACTER COUNTER
        .END    INIT                    ;END OF ASSEMBLY
```

Рис. 7.18. Программа чтения с перфоленты по прерываниям (PIC)

Фактически маска «*обезоруживает*» специфицированные прерывания и не разрешает прерывания только внутри группы, но не устанавливает какого-либо приоритета между прерываниями или группами устройств.

Ниже следует пример программы, демонстрирующей необходимость иметь приоритетность устройств. Эта программа ориентирована на телетайп и сетевой таймер машины PDP-11. После загрузки и запуска программа выводит на печать следующее сообщение:

```
WHAT TIME IS IT?
```

на которое пользователь отвечает четырехзначным числом. Впоследствии программа, отрабатывая прерывания от таймера каждые 1/60 секунды, следит за временем, отвечая

```
AT THE BELL THE TIME WILL BE: XX:XX:XX
```

каждый раз, когда вводится символ с клавиатуры.

Оба устройства в этом примере управляются по прерываниям. В то время как программа обработки прерываний с клавиатуры выводит сообщение о времени, тестируя бит вывода DONE, часы могут прервать ее при истечении очередной 1/60 секунды. Для этого приоритет таймера должен быть выше, чем клавиатуры *и* ее ISR печати времени, если мы хотим, чтобы «тики» * не терялись. (В том, что тики действительно могут быть потеряны, можно убедиться, изменив приоритет таймера в программе недалеко от метки NOFIX.)

Уровень приоритетности — это характеристика не только устройства. Все группы устройств имеют собственные уровни приоритетности; выполняющаяся программа ISR также имеет приоритет, равный *уровню приоритета CPU*. Каждое устройство, которому разрешено вызывать прерывания, имеет приоритет, вначале зафиксированный в программе обработки прерываний. Если, к примеру, этот приоритет равен 7 (т. е. он записан в PS программы обработки) и такое прерывание произошло, то никакая другая программа с PS, содержащим приоритет 7 или меньше, не сможет прервать текущую обработку. Из этого следует, что приоритет процессора, который указывается в PS, действует как маска прерываний от устройств I/O. Любая ISR, выполняющаяся на уровне приоритета CPU «n», может быть прервана только устройством, уровень приоритетности которого *больше, чем* «n».

Ниже приводятся адреса векторов прерываний и уровней приоритетности для терминала, устройства чтения/вывода на перфоленту и таймера машины PDP-11:

Устройство	Адрес вектора	Приоритет
Клавиатура терминала	60/62	4
Вывод терминала	64/66	4
Устройство чтения п/л	70/72	4
Устройство вывода п/л	74/76	4
Таймер	100/102	6

Рис. 7.19 и 7.20 демонстрируют примеры использования уровней приоритетности. Кроме того, программы используют рекурсии, прерывания и свободное обращение к подпрограммам, возможное благодаря стеку. Пример этот далек от тривиального. Рис. 7.19 раскрывает общую картину работы программы. Особый интерес

---

* Тик — это единица времени сетевого таймера. В зависимости от частоты сети, тик равен 1/60 или 1/50 секунды. — *Примеч. ред.*

представляет программа обработки прерываний от таймера, которая рекурсивно обращается к подпрограмме отсчета времени.

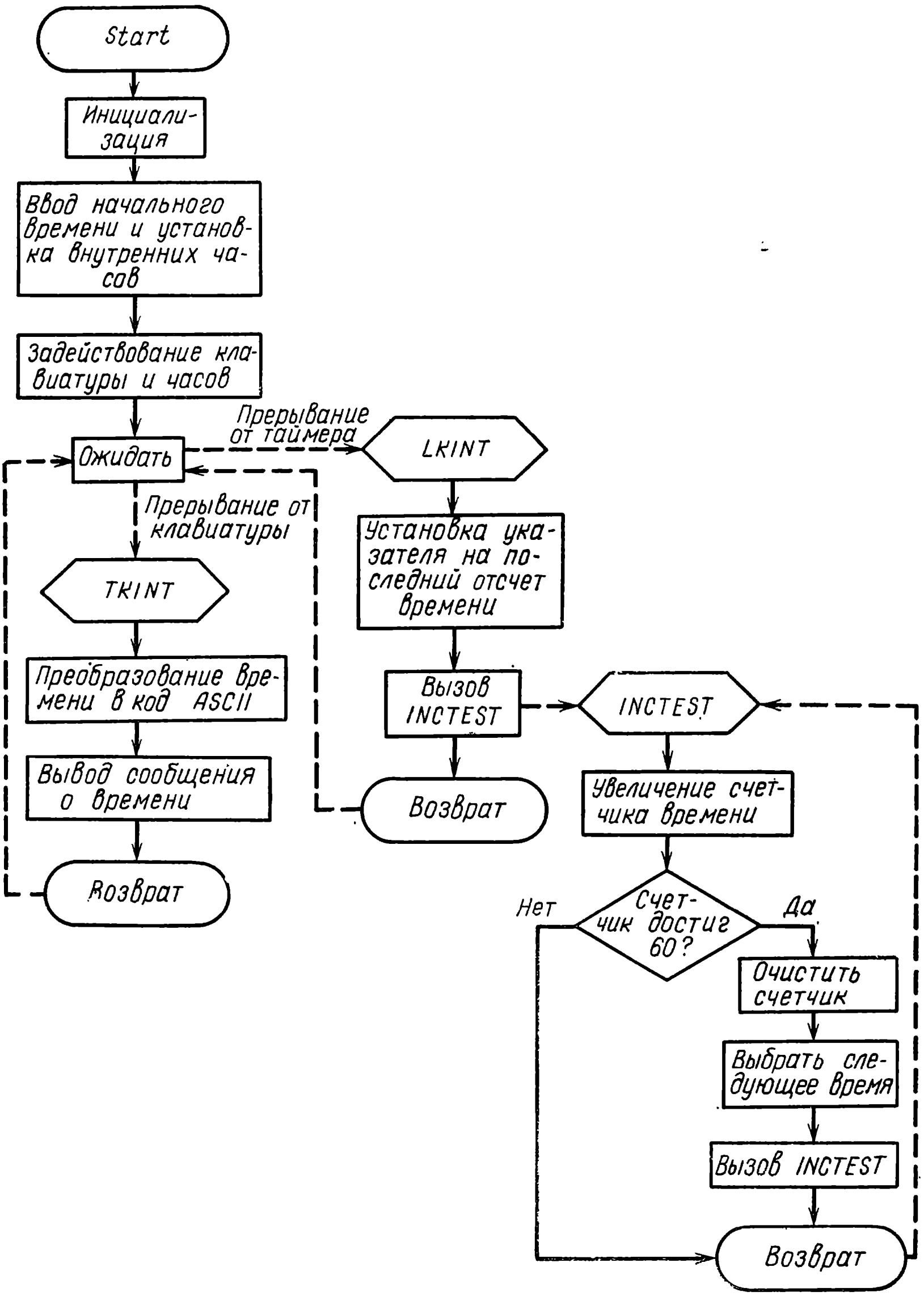


Рис. 7.19. Блок-схема «диктора» времени

Чтобы понять, как работает подпрограмма отсчета времени, надо анализировать стек после каждого обращения к ней. После того как выполнится предложение, следующее за меткой LKINT,

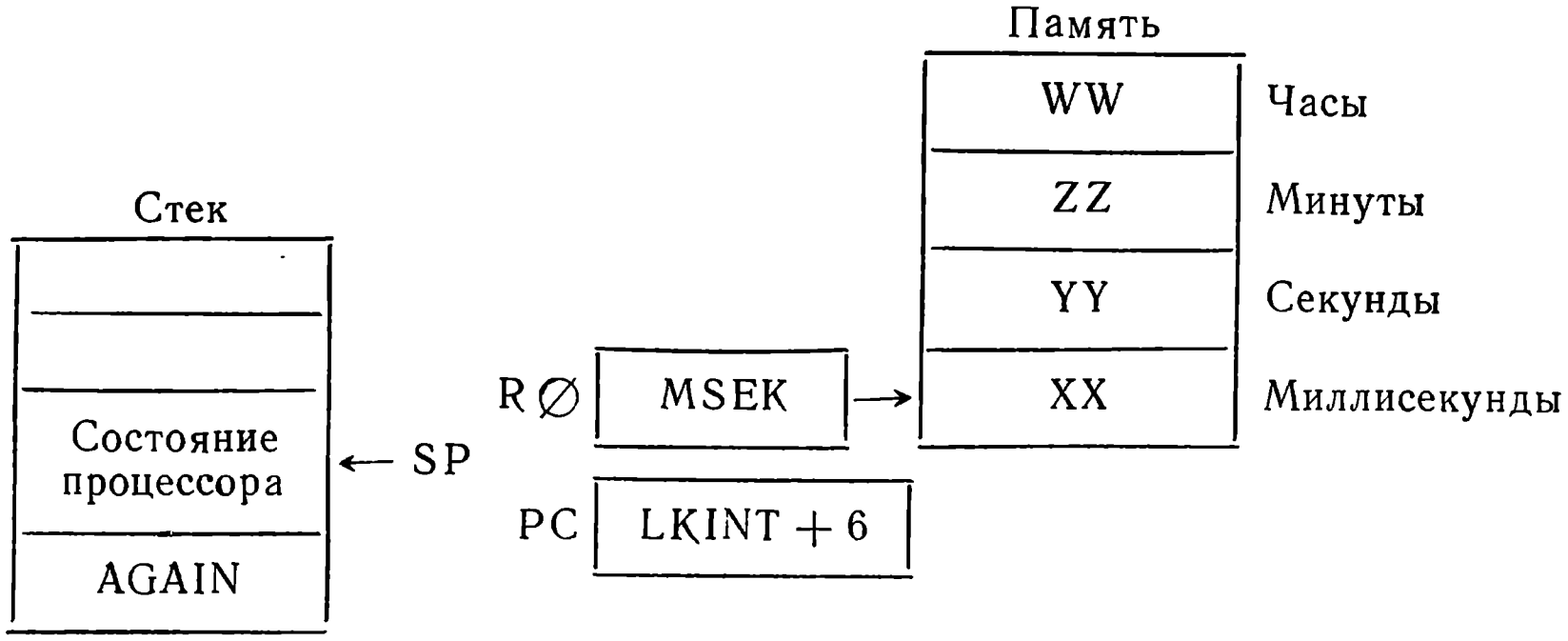
```
        TKCSR=177560
        TKDBR=177562
        TPCSR=177564
        TPDBR=177566
        LKCSR=177546
        ;
BEGIN:  MOV     #BEGIN,SP       ;INITIALIZE SP
        CLR     TKCSR           ;INITIALIZE KEYBOARD CSR
        CLR     TPCSR           ;INITIALIZE PRINTER CSR
;
        MOV     #M1,R2          ;ASK FOR
        MOV     #EM1,R3         ;THE TIME
        JSR     PC,DPR1         ;PRINT IT OUT
        MOV     #TIME,R2        ;ADDR OF TIME FIELD
        MOV     #4,R1           ;COUNT FOUR CHARACTERS
        MOV     R2,R3           ;PRINT ADDRESSES
NEXTD:  INC     TKCSR           ;READ THE TIME
TST1:   TSTB    TKCSR           ;TEST FOR A CHARACTER
        BPL     TST1            ;WAIT
        MOVB    TKDBR,(R2)      ;PUT IT IN TIME FIELD
        JSR     PC,DPR1         ;PRINT IT
        TSTB    (R3)+           ;NEXT BYTE
        SOB     R1,NEXTD        ;DECREASE COUNT, KEEP GOING
        MOVB    TIME+1,HOURS    ;LSD OF HOURS
        BIC     #177700,HOURS   ;CLEAR PARITY
        SUB     #60,HOURS       ;CONVERT TC OCTAL ('0' = 60 (8))
        BIC     #177700,TIME    ;CLEAR PARITY
NEXT:   CMPB    TIME,#61        ;ANY TENS?
        BLT     AROUND          ;NO
        ADD     #10.,HOURS      ;INCREASE VALUE BY 10.
        SUB     #1,TIME         ;DEC TENS COUNT
        BR      NEXT            ;ANYMORE?
AROUND: MOVB    TIME+3,MIN      ;GET MINUTES LSD
        BIC     #177700,MIN     ;CLEAR PARITY
        SUB     #60,MIN         ;CONVERT TO OCTAL ('0' = 60 (8))
        MOVB    TIME+2,R0       ;MUST CORRECT TENS
        BIC     #177700,R0      ;REMOVE PARITY
        MOV     #10.,R1         ;ADD 10 DECIMAL
        SUB     #61,R0          ;TEST FOR A ONE ('1' = 61 (8))
        BMI     NOFIX           ;NO TENS
        BEQ     ADD             ;ONE TEN
MORE:   ADD     #10.,R1         ;TRY AGAIN
        SOB     R0,MORE         ;COUNT THE TENS, MORE?
ADD:    ADD     R1,MIN          ;ADD IN # OF TENS
;
NOFIX:  MOV     #200,@#62       ;LEVEL 4 PRIORITY
        MOV     #TKINT,@#60     ;FOR THE TTY KBD ISR
        MOV     #340,@#102      ;LEVEL 7 PRIORITY
        MOV     #LKINT,@#100    ;FOR THE CLOCK ISR
        MOV     #101,TKCSR      ;INIT KBD
        MOV     #100,LKCSR      ;AND CLOCK
AGAIN:  WAIT                    ;NOTHING TO DO
        BR      AGAIN           ;HANG IN THERE
; CLOCK INTERRUPT SERVICE ROUTINE ******** CPU PRIORITY 7 ********
LKINT:  MOV     R0,R5           ;SAVE R0
        MOV     #MSEC,R0        ;ADDRESS OF LS FIELD
        JSR     PC,INCTEST      ;RECURSIVE CALL
        MOV     R5,R0           ;RESTORE R0
        RTI                     ;CLOCK UPDATED
```

Рис. 7.20.

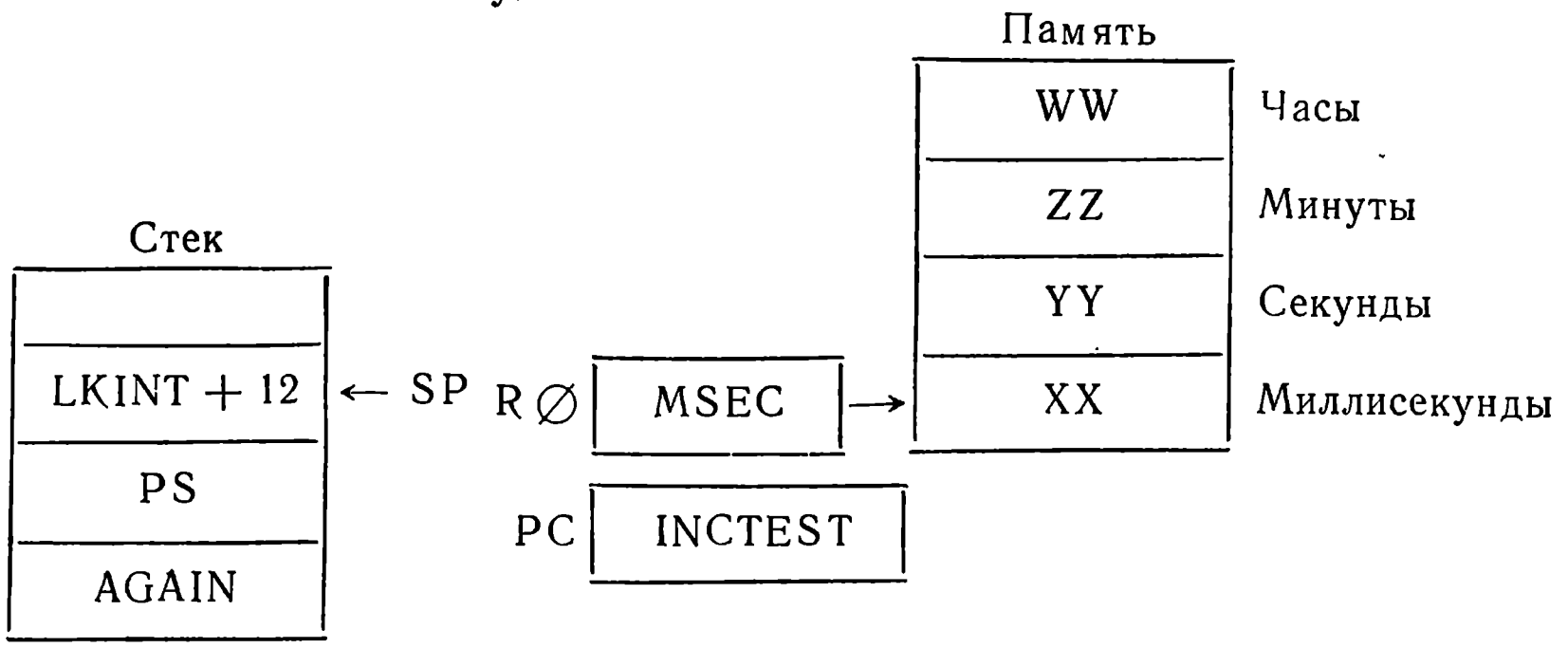
```
INCTEST:INC     (R0)            ;ADD ONE
        CMP     (R0),#60.       ;REACHED LIMIT?
        BNE     RETURN          ;NO
        CLR     (R0)            ;RESET FIELD
        TST     -(R0)           ;ADDR OF NEXT FIELD
        JSR     PC,INCTEST      ;CALL ME AGAIN
RETURN: RTS     PC              ;RETURN HOME
; TTY INTERRUPT SERVICE ROUTINE ********** CPU PRIORITY 4 ****
TKINT:  MOV     #M2,R2          ;PRINT OUT
        MOV     #EM2,R3         ;THE TIME
        JSR     PC,DPR1         ;MESSAGE
        MOV     #3,R3           ;NUMBER OF FIELDS
        MOV     #OUT,R2         ;OUTPUT AREA
        MOV     #HOURS,R4       ;FIRST FIELD ADDR
CNVRT:  CLR     R0              ;INITIALIZE
        MOV     (R4)+,R1        ;FIRST VALUE
LOOP:   CMP     R1,#10.         ;ANY TENS?
        BLT     ADDUP           ;NO
        INC     R0              ;YES, COUNT
        SUB     #10.,R1         ;DEC TENS
        BR      LOOP            ;DO IT AGAIN
ADDUP:  ADD     #60,R0          ;TENS IN ASCII
        MOVB    R0,(R2)+        ;STORE IT
        ADD     #60,R1          ;UNITS IN ASCII
        MOVB    R1,(R2)+        ;STORE IT
        TSTB    (R2)+           ;SKIP :
        SOB     R3,CNVRT        ;LOOP COUNT, DO IT THREE TIMES
        MOV     #OUT,R2         ;READY
        MOV     #BELL,R3        ;TO PRINT
        MOV     #101,TKCSR      ;RDR ENB
        TST     TKDBR           ;CLEAR DONE BIT
        JSR     PC,DPR1         ;YES
        RTI                     ;DONE AT LAST
; MESSAGE PRINTER ***************************************
DPR1:   CMP     R2,R3           ;ARE WE DONE?
        BHI     DPR2            ;YES
TST2:   TSTB    TPCSR           ;READY TO PRINT?
        BPL     TST2            ;NO
        MOVB    (R2)+,TPDBR     ;PUT IN BUFFER
        BR      DPR1            ;NEXT CHARACTER
DPR2:   RTS     PC              ;RETURN
; DATA **************************************************
M1:     .BYTE   15,12           ;CR AND LF
        .ASCII  /WHAT TIME IS IT?/
EM1:    .BYTE
        .EVEN
TIME:   .BYTE   0,0,0,0         ;FOUR CHAR TIME
OUT:    .BYTE   0,0,':,0,0,':,0,0
BELL:   .BYTE   7               ;STRIKE THE GONG
        .EVEN
HOURS:  .WORD   0
MIN:    .WORD   0
SEC:    .WORD   0
MSEC:   .WORD   0
M2:     .BYTE   15,12
        .ASCII  /AT THE BELL THE TIME WILL BE:.
EM2:    .BYTE   '
        .END    BEGIN
```

Рис. 7.20. Программа (NON PIC) «диктора» времени

содержание стека и R0 в символическом виде можно представить так:

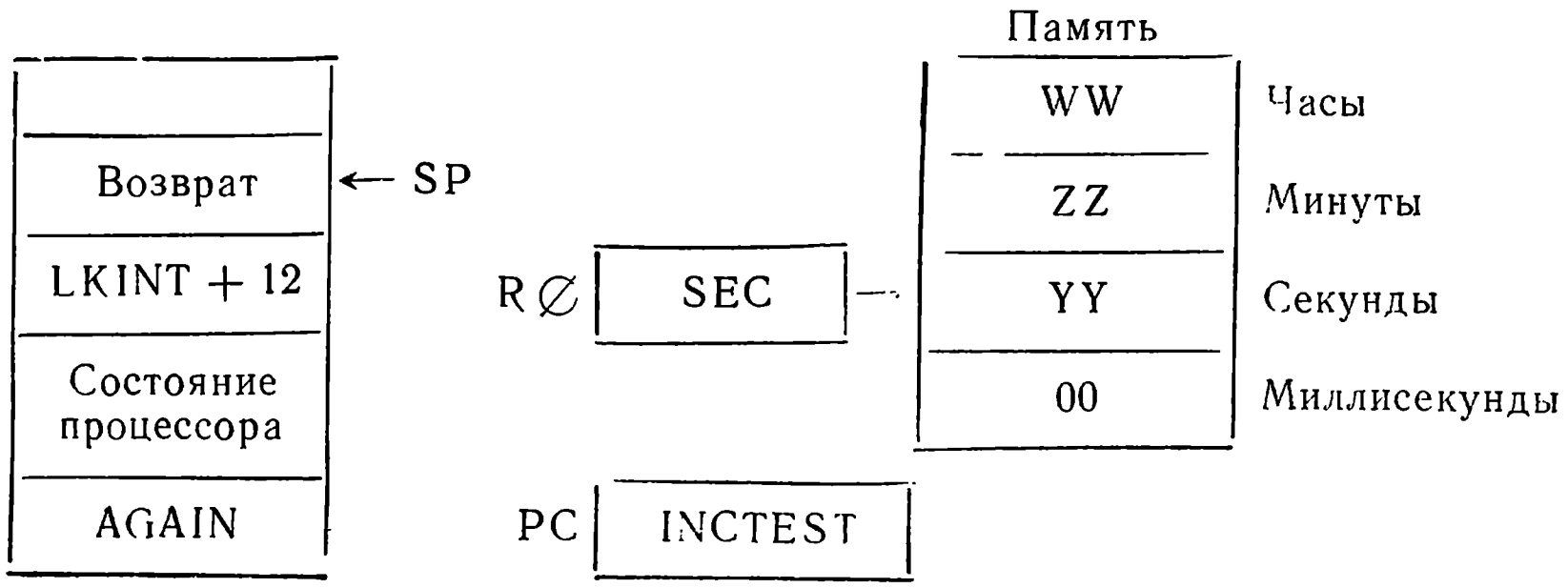


После того как будет вызвана подпрограмма INCTEST, картина в стеке изменится и будет такой:



Если значение в ячейке MSEC меньше чем 59, то оно будет увеличено на 1 и произойдет выход из подпрограммы, за которым последует выход из программы обработки прерывания с выталкиванием из стека PC и PS; далее программа снова будет ожидать очередного прерывания.

Предположим, однако, что величина в ячейке MSEC равна 59. В этом случае подпрограмма отсчета INCTEST установит значение MSEC = 0, продвинет в R0 указатель на ячейку SEC и рекурсивно вызовет саму себя. Картина в стеке будет следующая:



И снова проверяется ячейка памяти, на которую указывает R0. Если величина, находящаяся в ней, меньше 59, то повторится рассмотренное выше — это значение увеличится на 1 и произойдет выход из подпрограммы. А так как возврат влечет выполнение команды, следующей за вызовом, т. е. команды RTS PC, то вновь осуществляется выталкивание из стека. Процесс повторяется пока не произойдет возврат из прерывания. Далее программа ожидает нового прерывания. Фактически выполнение рекурсивных вызовов дело простое, ибо они выполняют только задачу восстановления стека.

При изменении уровня приоритетности программ обработки прерываний клавиатуры на 6 или более (например, MOV #300, @#62 в команде с меткой NOFIX) таймер будет не в состоянии прерывать ее в моменты вывода сообщений, и, следовательно, нарушится процесс отсчета времени. Это подчеркивает важность правильной установки приоритета CPU по отношению к приоритетам устройств.

7.5.8. Абсолютный и относительный режимы адресации при обращении к вектору прерываний и регистрам устройства

Проницательный читатель заметил, что в одних программах этой главы применяется абсолютный режим адресации при обращении к регистрам устройств и векторам прерываний, в то время как другие используют относительный режим. Необходимость применения абсолютной адресации в программах, записанных в PIC (например, рис. 7.2, 7.18), очевидна. Однако не совсем ясно, какой режим адресации предпочесть при написании программ в обычном (не PIC) коде (рис. 7.20).

Если для обращения к вектору прерываний или регистру устройства, обозначенных символами, используется абсолютный режим, а для прямой символической адресации локальных переменных, находящихся в памяти,— относительный, то спецификация соответствующего операнда *автоматически* становится позиционно-независимой. В этом смысле абсолютный режим является естественным (и, таким образом, более предпочтительным) для символической спецификации фиксированных адресов. Кроме того, хотя мнемоника со смысловой нагрузкой весьма полезна с точки зрения применения предпринимаемых программой действий, тем не менее применение абсолютного режима позволяет быть уверенным в том, что обращение происходит к фиксированной ячейке памяти. Так как при абсолютном режиме в ассемблерном листинге генерируются абсолютные адреса, можно визуально проверить значения вектора и регистра, не прибегая к проверке предложений, в которых эти значения присваиваются идентификаторам. И наконец, когда производится отладка с помощью интерактивной программы динамической отладки (DDP, разд. 8.3) по ассемблерному листингу и карте загрузки памяти (разд. 8.2), то применение абсолютного режима облегчает задачу изменения

различных ссылок на фиксированные ячейки, ибо программист меняет абсолютный адрес, а не смещение относительно PC.

Таким образом, относительная символическая адресация фиксированных ячеек памяти является *удовлетворительной* при использовании компоновщика, но абсолютная адресация *предпочтительнее* с точки зрения получения хорошей практики программирования.

7.6. ЗАПИСИ И БУФЕРИЗАЦИЯ

Хотя основные устройства ввода-вывода (телетайп, устройство чтения/записи на перфоленту) реализуют обмен символами с программой, тем не менее такие операции требуют специального

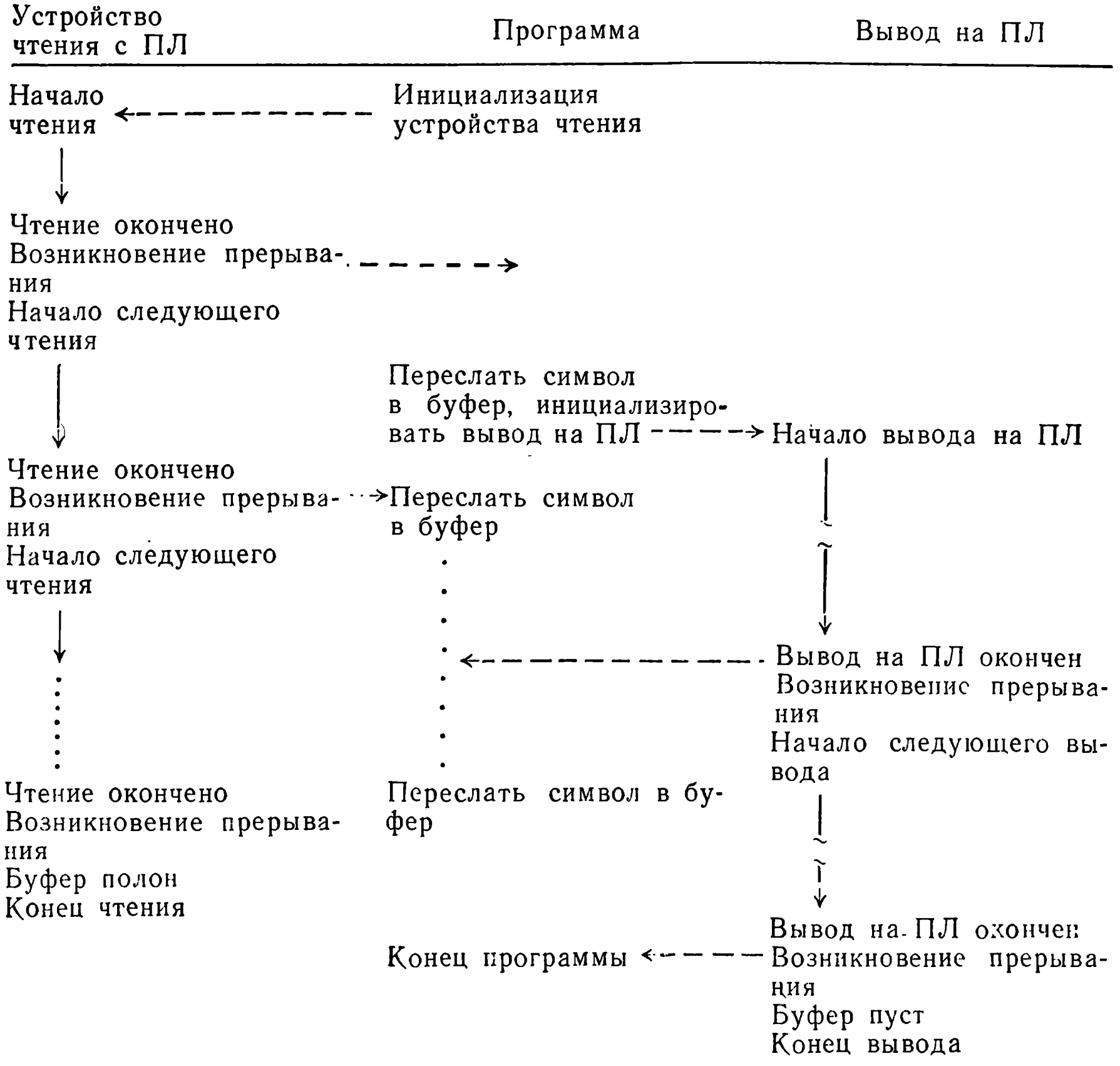


Рис. 7.21. Ввод-вывод с перекрытием

программирования. Это обусловлено тем, что вводимые или выводимые данные имеют вид наборов символов, таких, как десятичные числа, имена людей, восьмеричное представление слов памяти и строки ассемблерного кода. Другими словами, в операциях ввода-вывода участвуют *записи*, составленные из логически связанных символов.

Поскольку устройства не осуществляют формирование записей, на программисте лежит ответственность за блокировку символов при вводе и их деблокировку при выводе. Для обеспечения совместимости скоростей передачи потока записей между программой и устройством служат *буферы* — непрерывные участки памяти, в которых временно размещаются данные. Буферы имеют очень важное значение при обмене информацией с устройствами, имеющими различные скорости. Так, например, если чтение с перфоленты выполняется в шесть раз быстрее, чем вывод из нее, то буфер должен обеспечивать одновременность операций (разумеется, ввод начинается раньше вывода).

Перекрытие операций ввода и вывода устройства чтения и записи на перфоленту показано на рис. 7.21. Каждое устройство работает с максимальной скоростью передачи в режиме прерываний. Так как устройство чтения с перфоленты имеет скорость передачи выше, оно первым и завершит обмен. После ввода данных в буфер программа выводит их на перфоленту и заканчивает свою работу.

7.6.1. Совмещение вычислений и операций ввода-вывода

Кроме того, буферы необходимы для совмещения процесса вычислений с операциями ввода-вывода. Обычно для этой цели применяют два буфера: пока один заполняется (или опустошается), другой обрабатывается программой, решающей задачу. В общем случае число буферов зависит от их использования программой и устройствами, а также от размеров и может быть больше двух. В *сбалансированной системе* количество и объемы буферов таковы, что обеспечивают полное совмещение вычислений и операций ввода-вывода. Если CPU занят менее 100% времени, то система *ограничена вводом-выводом.* В том случае, когда устройство ввода-вывода загружены не на 100%, говорят, что система *ограничена вычислениями.*

Та или иная степень совмещения вычислений и ввода-вывода зависит от многих факторов, включая такие, как конфигурация ЭВМ, вопросы экономичности, загруженность системы и т. д. Всеми этими вопросами ведает системный программист, отвечающий за проектирование и реализацию операционной системы.

7.7. СИСТЕМЫ ПРОГРАММИРОВАНИЯ ВВОДА-ВЫВОДА (IOPS)

Для того чтобы эффективнее использовать устройства ввода-вывода и помочь пользователю в кодировании соответствующих программ, разработчики машин снабжают их *системой программирования I/O* (IOPS) *. Такая система

1) освобождает программиста от необходимости глубокого знания различных деталей, связанных с непосредственным взаимодействием с внешним устройством;

---

* Обычно их называют системами управления ввода-вывода. — *Примеч. ред.*

2) обеспечивает лучший сервис и организацию обмена данными;
3) облегчает программирование процедур ввода-вывода благодаря макрокомандам, записанным на языке Ассемблера;
4) обеспечивает функционирование программ в различных вычислительных средах.

В дополнение программист может применить IOPS для

1) обслуживания асинхронных процессов обмена;
2) организации перекрывающихся процессов ввода-вывода;
3) создания программ, не зависимых от устройства;
4) блокирования и буферизации.

Так, в операционной системе RT-11 системные макрокоманды, обеспечивающие программирование процедур ввода-вывода, делятся на три категории. Первая включает *команды передачи данных*, такие, как READ и WRITE. Вторую группу образуют *команды инициализации* файлов и получения информации об их состоянии, такие, как OPEN, CLOSE и др. Последняя, третья, категория представляет собой средства управления, такие, как WAIT и RESTART.

7.7.1. Пример IOPS, работающей в операционной системе RT-11

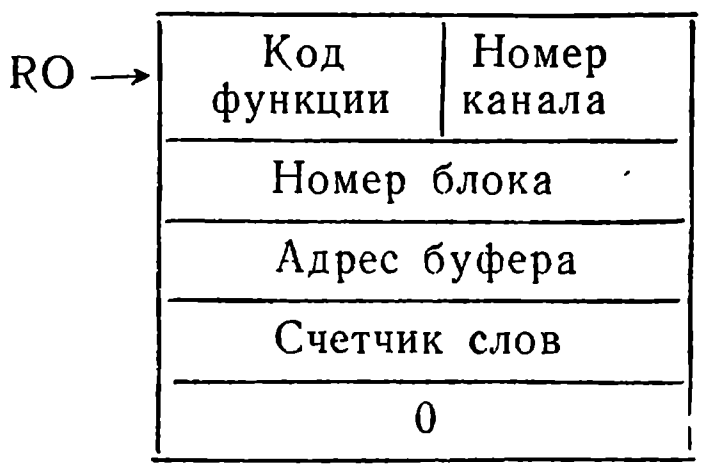
Рассмотрим пример использования IOPS для создания программной последовательности ввод-обработка-вывод. Пример включает

1) спецификацию необходимых буферов и управляющих блоков;
2) загрузку драйверов необходимых устройств;
3) чтение в буфер и ожидание окончания ввода;
4) проверку правильности чтения;
5) обработку только что прочитанных данных;
6) вывод из буфера.

В сокращенном виде такая программная последовательность на базе программных запросов может быть представлена так:

```
START:  .FETCH  #HNDLSP,#PRNAME                 ;GET PAPER TAPE HANDLER
                .
                .
        .READW  #AREA,#CHN0,#BUFF,#WDCT,#BLK    ;READ DATA AND WAIT
        BCS     RDERR                           ;CARRY SET IF ERROR
                .
                .
        PROCESS BUFFER
                .
                .
        .WRITW  #AREA,#CHN1,#BUFF,#WDCT,#BLK    ;WRITE DATA AND WAIT
        BCS     WDERR                           ;CARRY SET IF ERROR
                .
                .
PRNAME: .RAD50 /PR/                             ;NO FILE NAME NEEDED
        .WORD 0                                 ;USE "0"
AREA:   .BLKW   10                              ;PROGRAM REQUEST BLOCK
BUFF:   .BLKW   80.                             ;DATA BUFFER
CHN0=0                                          ;CHANNEL 0
CHN1=1                                          ;CHANNEL 1
WDCT=80.                                        ;NUMBER OF DATA WORDS
BLK=0.                                          ;BLOCK TO BE READ/WRITE
HNDLSP=.                                        ;HANDER GOES HERE
```

Здесь демонстрируются специфические черты и требования операционной системы RT-11: во-первых, то, что драйвер должен загружаться и размещаться вместе с программой пользователя; во-вторых, то, что некоторые запросы должны включать область, где размещаются аргументы макровызовов. Адрес этой области хранится в R0. Так, для программного запроса .READW ее содержимое имеет следующий вид:



Эта область служит для указания кода выполняемой функции, номера логического канала, по которому будет выполняться операция, логического блока, с которого начинается ввод, адреса буфера и количества передаваемых блоков.

Читатель вправе задать вопрос: «А почему не разместить все это в стеке?». Такое решение возможно, но разработчики RT пошли по другому пути *.

Как следует из приведенного фрагмента программы, бит переноса сигнализирует об ошибке и, следовательно, требует наличия команды BCS. Дальнейшая обработка определяет вид ошибки. Буква «W» в запросах READW и WRITEW означает «ждать» окончания операции. Другие формы READ/WRITE не включают этой функции и соответствующие программы не переходят в состояние ожидания. Эти формы применяются тогда, когда необходимо совместить процесс ввода-вывода с вычислениями. Следующая программа иллюстрирует применение простых системных макрокоманд ввода-вывода для работы с консолью.

```
SYSTST: .PRINT  #MESS1          ;PRINT MESSAGE #1
        MOV     #BUFF,R1        ;SET POINTER TO INPUT BUFFER
        CLR     R2              ;RESET CHARACTER COUNTER
LOOP1:  .TTYIN  (R1)            ;GET CHARACTER
        INC     R2              ;INCREMENT COUNTER
        CMPB    #12,(R1)+       ;END OF INPUT? (LINE FEED?)
        BNE     LOOP1           ;NO!
        MOV     #BUFF,R1        ;YES, RESET BUFFER POINTER
        .PRINT  #MESS2          ;OUTPUT RESPONSE
LOOP2:  .TTYOUT (R1)+           ;OUTPUT NAME
        SOB     R2,LOOP2
        .EXIT
MESS1:  .ASCII  /WHAT IS YOUR NAME?/<200>
MESS2:  .ASCII  /HELLO THERE, /<200>
BUFF:   .BLKB   80.
        .END    SYSTST
```

* Использование стека для заполнения параметров макровызовов применяется, например, в другой стандартной для ЭВМ PDP-11 операционной системе — RSX-11. — *Примеч. ред.*

Использование <200> в MESS1 и MESS2 позволяет заканчивать сообщение, не применяя ⟨CR⟩ и ⟨LF⟩. Угловые скобки служат также для указания того, что выражение в них должно употребляться как единый терм. И последнее — конструкция ⟨ ⟩ позволяет указать ассемблеру восьмеричный код непечатаемых символов, таких, как ⟨CR⟩, ⟨LF⟩ и tab.

Макрокоманды служат для обращения к системе RT-11. В ответ на них операционная система выполняет затребованные пользователем функции.

7.7.2. Проблема связи с IOPS

В больших вычислительных системах для того, чтобы воспользоваться возможностями, предоставляемыми макрокомандами, необходимы как макроассемблер, так и связывающий загрузчик *. Поскольку в базовом программном обеспечении мини-компьютеров макроассемблер или загрузчик может отсутствовать (или включаться как самостоятельные компоненты при дальнейшем развитии системы), на программиста возложено создание расширений макровызовов в операторы ассемблера и соответствующая привязка программы к IOPS. Даже в таких развитых операционных системах, как RT-11, включающих ассемблер и загрузчик, все равно остается проблема привязки программы к IOPS. Причина состоит в том, что драйверы устройств в операционных системах мини-ЭВМ обычно не резидентны. И только в период выполнения программы драйвер загружается и осуществляется его взаимодействие с программой в соответствии с выданной в ней макрокомандой.

Поэтому наиболее часто применяемый способ связи программы с IOPS — это использование машинных команд, называемых *SVC* (service calls) или *EMT* (emulator traps), вызывающих прерывания. Такие прерывания, инициируемые программой, обслуживаются системой подобно прерываниям по вводу-выводу. В результате текущее PSW заменяется новым, указывающим на программы управления вводом-выводом. При выполнении SVC или EMT допустимы операции, характерные для вызова подпрограмм, в том числе и передача аргументов (старое PC в PSW указывает на первое слово в списке аргументов). Кроме того, организуемые при помощи этих двух команд взаимодействия пользовательских программ и программ IOPS удовлетворяют общей схеме функционирования системы. Здесь в первую очередь стоит отметить следующие особенности программ управления вводом-выводом:

1) в них не нужно сохранять состояние процессора (за исключением используемых регистров);

---

* В современных операционных системах связывание программ выполняет редактор связей (построитель задач, компоновщик и т. д.), а не загрузчик. — *Примеч. ред.*

2) работа этих программ возможна на любом желаемом уровне приоритета;
3) они обеспечивают прямую связь между программой пользователя и IOPS через фиксированную ячейку памяти.

Последний пункт наиболее важен. Команда EMT позволяет прямо связать программу пользователя и IOPS, без помощи компоновщика или загрузчика. Для этого необходимо только, чтобы адрес IOPS был предварительно загружен в вектор прерываний, соответствующий EMT. Поскольку обращение в IOPS может быть вызвано различными причинами, определение конкретного вызова осуществляется по операнду EMT. В этом отношении обработка прерывания по EMT происходит аналогично тому, как это делается в одноуровневой системе прерываний, рассмотренной ранее.

7.7.3. Другие прерывания

Затратим теперь некоторое время на рассмотрение прерываний, не связанных с операциями ввода-вывода, поскольку такие прерывания могут быть использованы для выполнения различных функций, в частности, для
1) индикации сбоев в программе: ошибка адресации, неправильная команда, анормальный результат выполнения арифметической операции (например, деление на нуль) и др.;
2) обработки ошибок машины, таких, как обращение по нечетному адресу или автоматически выявленный сбой оборудования;
3) установления флага в случае сбоя питания или возникновения прерывания от клавиатуры консоли.

Поскольку эти прерывания изменяют текущее PSW, их можно использовать и для изменения *состояния защиты* системы (см. разд. 7.8).

Так, например, некоторые команды ввода-вывода вызовут прерывания, если система находится в пользовательском режиме. С другой стороны, в привилегированном состоянии, или, иначе, в режиме монитора, никаких прерываний не произойдет. Следовательно, все требования ввода-вывода должны управляться системой программирования I/O, которая активизируется командами SVC и EMT. Использование этих команд приводит к изменению *состояния защиты* ЭВМ — переходу от пользовательского режима (защищенного состояния) к системному (незащищенному состоянию). От состояния защиты вычислительной системы зависит адресуемая область памяти, а также возможность выполнения отдельных команд. Так как перевод машины в состояние «система» производится путем установки бита в PSW, это можно делать автоматически, гарантируя, таким образом, программе обработки прерываний доступ ко всем ресурсам ЭВМ. Концепция «состояние системы» будет более полно изложена в гл. 10, где

обсуждается более современная операционная система (многопрограммная и с разделением времени).

7.7.4. Обслуживание прерываний системой

Представляется полезным провести краткий анализ обращения к системе IOPS на PDP-11. С этой целью мы снова рассмотрим, как расширяется макрокоманда .READW.

Пусть макровызов имеет следующий формат:

```
.READW  #AREA,#CHN0,#BUFF,#WDCT,#BLK
```

Его макрорасширение будет выглядеть так:

```
MOV     #AREA,R0
MOVB    #8.,1(R0)
MOVB    #CHN0,(R0)
MOV     #BLK,2(R0)
MOV     #BUFF,4(R0)
MOV     #WDCT,6(R0)
MOV     #0,8.(R0)
```

Очевидно, что для выполнения эта макрокоманда требует некоторой области памяти, а также команды EMT, которая обеспечит передачу управления монитору RT-11 для обработки запроса ввода-вывода.

7.7.5. Пример сопрограммы, применяющей макрокоманды системы RT-11

В гл. 4 было упомянуто, что сопрограммы используются в процессах ввода-вывода и представляют собой одну из основных операций, выполняемых современной операционной системой. Пример, который приводится ниже, демонстрирует их применение при программировании процедур ввода-вывода с двумя буферами и с совмещением с вычислениями.

```
Write O1                              Write O2
Read I1          concurrently         Read I2  concurrently
Process I2                            Process I1
```

Здесь O1 и O2 обозначают буферы вывода 1 и 2, а I1 и I2 буферы ввода 1 и 2. Напомним, что команда JSR PC, @(SP)+ всегда выполняет переход по адресу, занесенному в вершине стека, и заменяет его новым адресом возврата. Таким образом, при выполнении команды JSR, обозначенной меткой B, происходит переход на разные ячейки: сначала на A, затем на ячейку, идущую после JSR. Все остальные команды JSR выполняют переход на B + 2 (рис. 7.22). Здесь подразумевается также, что время обработки данных превышает время их чтения/записи.

Хотя эта программа короткая, она показательна в плане мощного и элегантного программирования процедур ввода-вывода

для двух буферов, причем здесь обмен данными совмещается с вычислениями. Она демонстрирует также силу и «способности» малой машины, позволяющей создавать системы реального времени и разделения времени, а также телекоммуникационные системы.

```
BEGIN:                                          ;INITIALIZATION
        .
        .READW  #AREA1,#CHN1,#I1,....,          ;READ INTO I1 TO START PROCESS,
        .                                       ;       WAIT FOR READ COMPLETE
        MOV     #A,-(SP)                        ;INITIALIZE STACK FOR FIRST JSR
;
B:      JSR     PC,@(SP)+                       ;DO I/O FOR O1 AND I1, OR O2 AND I2
        .
        PERFORM PROCESSING: I1 → O1 OR I2 → O2
        .
        BR      B                               ;MORE I/O
;
;I/O CO-ROUTINES FOLLOW
;
A:      .READ   #AREA1,#CHN1,#I2,...            ;INTITIATE READ INTO I2
        .
        SET PARAMETERS TO PROCESS I1 TO GET O1
        .
        JSR     PC,@(SP)+                       ;GO TO B+2
;
        .WRITE  #AREA2,#CHN2,#O1,...            ;INITIATE WRITE FROM O1
        .READ   #AREA1,#CHN1,#I1,...            ;INITIATE READ INTO I1
        SET PARAMETERS TO PROCESS I2 TO GET O2
        .
        JSR     PC,@(SP)+                       ;GO TO B+2
;
        .WRITE  #AREA2,#CHN2,#O2,...            ;INITIATE WRITE FROM O2
;
        BR      A                               ;GO TO A
        .END    BEGIN
```

Рис. 7.22. Пример сопрограммы: макросы системы RT-11, обеспечивающие перекрытие обработки и ввода-вывода

7.8. ДИСПЕТЧЕР ПАМЯТИ И PSW

Раньше мы рассматривали отображение памяти как свойство, позволяющее расширить адресацию в системах, имеющих более 28K физической памяти. Однако диспетчер памяти не только расширяет возможности адресации. В совокупности с битами PSW и некоторыми специальными командами он может обеспечить работу CPU в трех режимах: ядра, пользовательском и супервизора. Причем для каждого режима имеется свой указатель стека. Режим супервизора реализован только на моделях 11/45, 55, 70.

7.8.1. Содержимое полей PSW

Как мы видели, в младшем байте PSW находятся коды условий (N, Z, V и C), а также текущее значение приоритета CPU. Младший байт содержит также T-бит.

Старший байт PSW содержит поля бит *текущего* и *предыдущего* режимов процессора [14 : 15] и [12 : 13] соответственно. В 11/45, 55 и 70, когда установлен 11-й бит, система переключается на второе множество регистров (только R0 ÷ R5). Так как

в программном обеспечении фирмы DEC эти регистры никогда не употребляются, пользователь может постоянно применять их в качестве указателей памяти в *отдельных* программах обработки прерываний с весьма большой скоростью обработки. Если бит 11 установлен в векторе прерываний программы ISR, то последняя автоматически обращается ко второму множеству регистров, не заботясь о их сохранении/восстановлении. При скорости анализа прерываний, равной 10kHz и более, возможной в системе реального времени, использование второго множества регистров R0 ÷ R5 в ISR позволяет сохранить время, эквивалентное мощности целого CPU машины 11/34!

7.8.2. Диспетчер памяти и режимы процессора

Сейчас мы покажем, как диспетчер памяти вычисляет физические адреса. Поскольку не все программы требуют полной страницы памяти (равной 4K), поле длины страницы в регистре описания страницы (PDR) позволяет указывать переменную длину, от 1 до 128 32-словных блоков. Начальный адрес каждой страницы — величина, кратная 32 словам, а ее максимальная длина — 4096 слов. Страницы могут размещаться в памяти произвольно (как это было показано на рис. 3.5).

Формирование физического адреса иллюстрируется на рис. 7.23

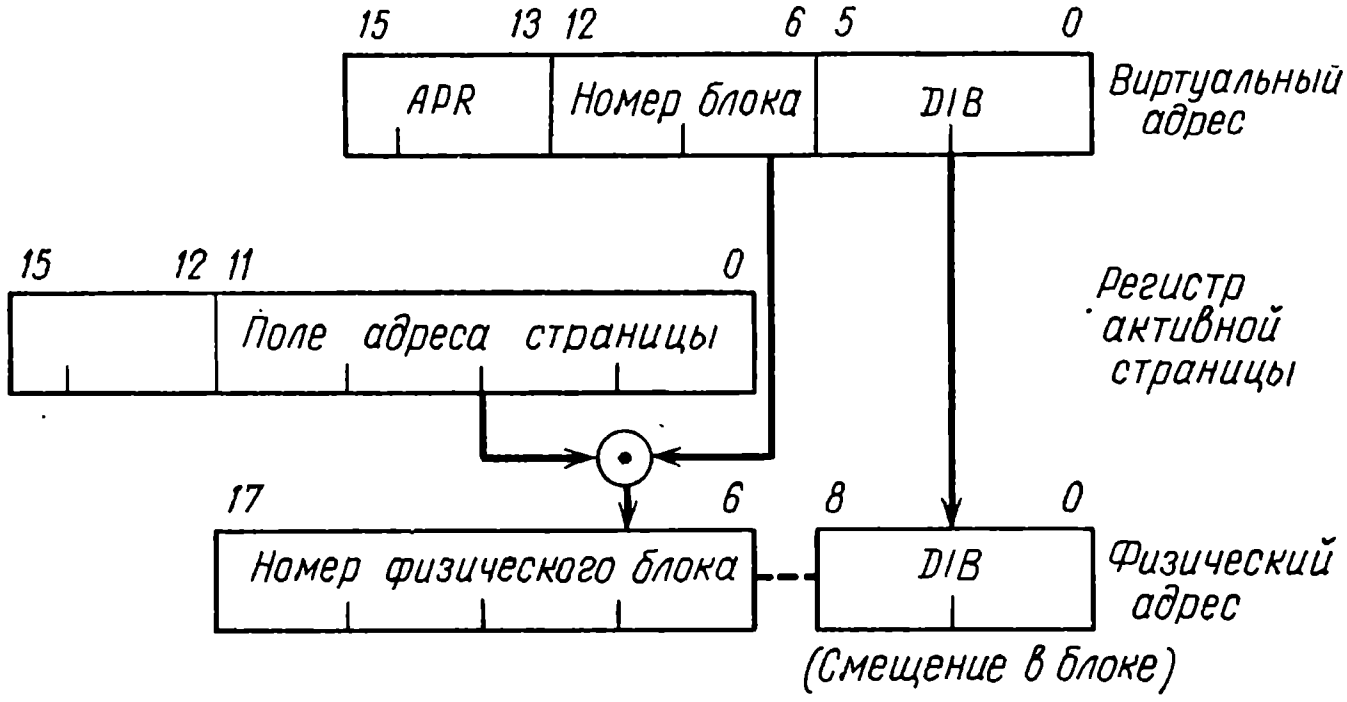


Рис. 7.23. Построение физического адреса

Последовательность построения физического адреса:

1. Выбирается множество регистров активных страниц, определяемое текущим режимом, установленным в PSW.
2. По значению виртуального адреса определяется один из регистров активных страниц (APR0 ÷ APR7).
3. Поле адреса страницы, выбранного регистра активной страницы, содержит ее начальный адрес в виде номера блока в физической памяти.
4. Номер блока из виртуального адреса складывается с номером блока из поля адреса страницы, а результат дает номер блока в физической памяти, в котором находится требуемая ячейка.

5. Смещение в блоке из поля смещения виртуального адреса добавляется к номеру физического блока памяти, образуя 18-битовый физический адрес.

Остановимся подробнее на особенностях отображения памяти. Во-первых, процессор воспринимает 32К-словное адресное пространство программы как один непрерывный блок памяти, в то время как ее страницы могут располагаться в несмежных участках физического адресного пространства. Во-вторых, в зависимости от диапазона виртуальных адресов и содержимого соответствующих регистров диспетчера памяти, страницы виртуального адресного пространства программы могут отображаться как в младшие, так и в старшие адреса физической памяти. В-третьих,

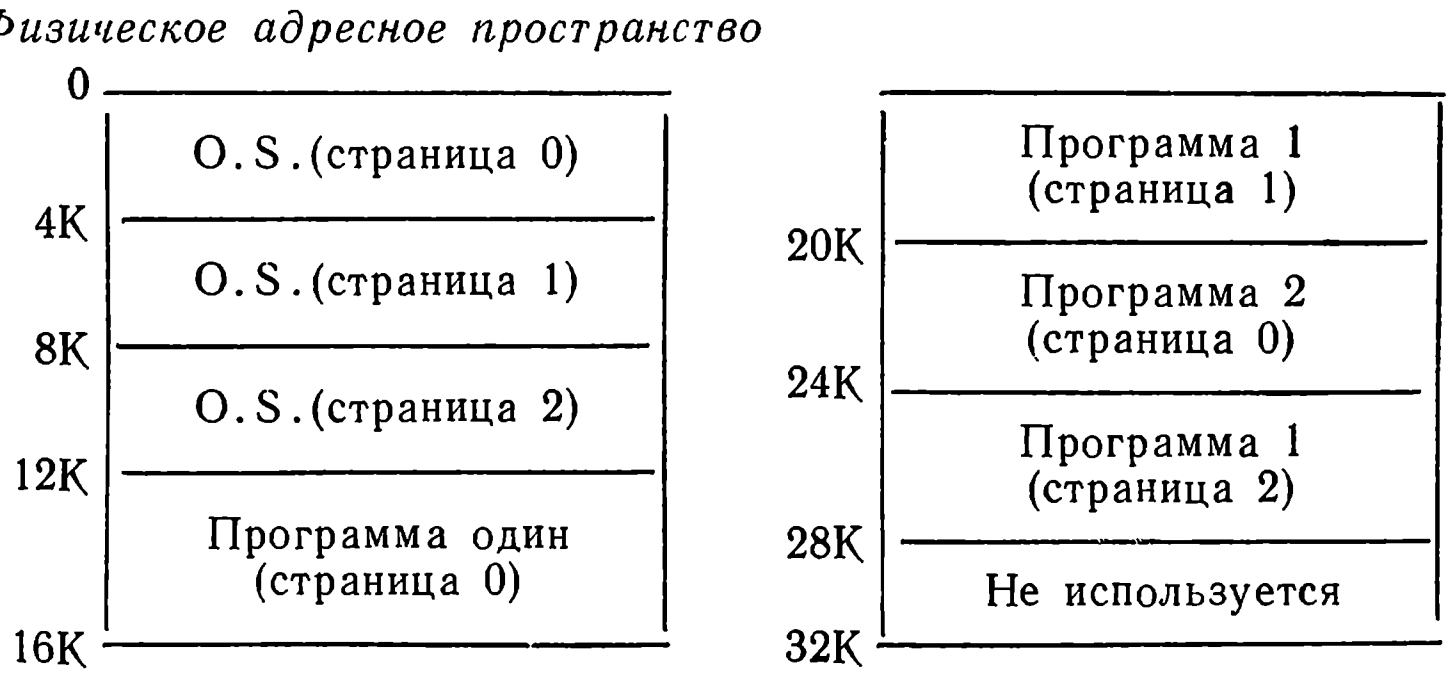


Рис. 7.24. Распределение памяти

все страницы начинаются на границах 32-словных блоков *. Наконец, в-четвертых, страницы отображаются независимо, и две или более из них могут адресовать один и тот же участок физической памяти. Используя более чем один регистр диспетчера памяти (из адресованного множества регистров), можно организовать доступ к одному и тому же участку памяти из разных мест одной или нескольких программ. При этом допустимо задание различных прав доступа (чтение, запись и т. д.).

Имеется несколько дополнительных команд, которые помогают программисту использовать диспетчер памяти. Они обеспечивают связь между пользователем и ядром системы. Так, если операционной системе надо получить данные из пространства, распределенного для пользователя, то выполняется команда MFPD, которая загрузит данные с помощью регистров APR предыдущего (пользовательского) режима.

Применение диспетчера памяти демонстрируется на примере. Предположим, что память распределена так, как это показано на

* Кроме того, длина страницы также измеряется в 32-словных блоках. Это обусловлено необходимостью оперирования 18-битовым адресом при 16-битовом слове ЭВМ. Так, если программный раздел в операционной системе RSX-11 начинается с физического адреса 600 000 (в байтах), то в 32-словных блоках его адрес будет 6 000, т. е. уместится в 16 разрядов. — *Примеч. ред.*

рис. 7.24, где первые три страницы физической памяти содержат операционную систему, следующие две — первые 8К программы 1, шестая страница — программу 2 и в последней физической странице находится остаток программы 1. (Мы предполагаем, ради простоты изложения, что все программы требуют одной или более страниц по 4К.) Чтобы разместить операционную систему, необходимо загрузить регистры PAR и PDR значениями смещения и управляющей информацией, а затем включить диспетчер памяти. Последовательность команд будет следующая:

```
KPAR0=772340                ;KERNEL MEMORY MANAGEMENT REGISTER
KPAR1=772342                ;LOCATED IN THE I/O PAGE
KPAR3=772344                ;GIVING THE RELOCATION VALUE 100
KPDR0=772300                ;KERNEL PAGE DESCRIPTOR REGISTER
          .
          .
UPAR0=777640                ;USER PAR
          .
          .
UPDR3=777606                ;USER PDR
          .
SR0=777572                  ;MEMORY MANAGEMENT CONTROL REGISTER
  .
  .
MOV       #0,@#KPAR0        ;PAGE ZERO STARTS AT PHYSICAL ADDRS ZERO
MOV       #200,@#KPAR1      ;PAGE ONE FOLLOWS AT PHYSICAL ADDRS 20000
MOV       #400,@#KPAR2      ;AND SO ON

MOV       #77406,@#KPDR0    ;4K PAGES TO BE READ/WRITE
MOV       #77406,@#KPDR1    ;
MOV       #77406,@#KPDR2    ;

MOV       #1,@#SR0          ;TURN ON MEMORY MANAGEMENT
  .
  .
```

После этого операционная система передает управление пользователю:

```
;
    MOV       #600,@#UPAR0      ;PAGE ZERO STARTS AT 60000
    MOV       #1000,@#UPAR1     ;NEXT PAGE IS CONTIGUOUS
    MOV       #1400,@#UPAR2     ;BUT LAST PAGE IS NOT
;
    MOV       #77406,@#UPDR0    ;4K PAGES TO BE READ/WRITE
    MOV       #77406,@#UPDR1    ;
    MOV       #77406,@#UPDR2    ;
;
    MOV       #140000,-(SP)     ;PUSH USER MODE INTO STACK
    CLR       -(SP)             ;START AT VIRTUAL ADDRESS ZERO
    RTI                         ;POP STACK AND RETURN
```

Таким образом, регистры диспетчера памяти применяются так, что 12К адресного пространства заняты операционной системой, причем ее виртуальные адреса совпадают с физическими. Далее система устанавливает регистры диспетчера памяти 1-й программы для отображения ее виртуальных адресов от 0 до 12К на физические (не непрерывные) от 12К до 20К и от 24К до 28К.

Чтобы программа пользователя начала работать, система помещает в стек ее PS и PC, затем по команде RTI эти два слова восстанавливаются с установленным в PSW режимом пользователя и программа начинает выполняться с виртуального адреса 0.

В действительности описанные программы не смогут успешно выполняться. Читатель заметит, что операционная система не инициализировала APR страницы ввода-вывода. Поэтому при включении диспетчера памяти путем установки регистра управления SR0 достижимыми адресами для ядра будут только адреса от 0 до 12K. Адреса страницы I/O будут вызывать сбои в диспетчере памяти, если операционная система попытается обратиться к ним при работе с программой 1.

УПРАЖНЕНИЯ

1. Напишите программу вывода сообщения «HELLO?» на терминал.

2. Напишите стандартную подпрограмму для терминала, обеспечивающую вывод символов табуляции. При входе в нее число символов помещено в R0.

3. Напишите программу чтения информации с перфоленты и вывода ее на перфоленту восьмеричными цифрами, по три в группе.

4. Напишите подпрограмму, принимающую от 1 до 6 цифр с телетайпа и формирующую 16-битовое слово в R0. Каждый введенный символ должен повторяться на телетайпе как «эхо». Предположите также, что строка заканчивается возвратом каретки, а наша программа вставляет перевод строки.

5. Выполните упр. 3, используя прерывания.

6. Напишите программу для чтения 400 символов с устройства чтения с перфоленты и одновременно вывода первых 100 из них на перфоленту и печать. Будьте осторожны при прекращении чтения, ибо медленно работающие печать и вывод на перфоленту могут не успеть выполнить свои операции.

7. Придумайте схему для измерения времени выполнения программы. Эта схема должна работать с точностью до 16,6 миллисекунды.

8. Запишите упр. 3, применяя макрокоманды системы RT-11.

9. Можно ли выполнить упр. 6, используя макрокоманды системы RT-11?

10. Закодируйте сопрограмму, находящуюся на странице 198 и работающую с двумя буферами так, чтобы она получила ввод с медленно работающего устройства чтения с перфоленты, а выводила на перфоленточное устройство с большой скоростью пробивки.

БИБЛИОГРАФИЯ

Программирование I/O является весьма специфической областью в том смысле, что у каждого типа ЭВМ свои собственные команды, а следовательно, и свои специфические черты программирования. В книгах Flores I. (1969), Hellerman H. (1967) и Foster C. C. (1970) I/O обсуждается с концептуальной точки зрения. В других книгах, подобных этим, I/O рассматривается применительно к особенностям машины. Что касается PDP-11, то наилучшим источником является Peripherals Handbook, где рассматриваются не только периферийные устройства, но и расширение UNIBUS, интерфейсы связи, а также возможности управления и получения данных.

Глава 8

СИСТЕМНЫЕ СРЕДСТВА РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

В наши дни все вычислительные машины от самых малых моделей мини-ЭВМ до огромных «пожирателей чисел» снабжаются развитыми пакетами системного программного обеспечения. В состав этих пакетов входят программы и соответствующая документация, позволяющие программисту писать, редактировать, ассемблировать, компилировать, отлаживать и выполнять программы, используя все возможности ЭВМ.

Системное программное обеспечение непрерывно совершенствуется системными программистами с тем, чтобы сделать доступ к ЭВМ легче и уменьшить затраты времени на это. Большинство операционных систем имеет открытую модульную структуру, что позволяет пользователю приспособить ее для конкретных условий. С этой позиции операционную систему можно рассматривать как интерфейс между характеристиками аппаратуры и потребностями пользователя.

В предыдущих главах читатель познакомился с тремя составными частями системного программного обеспечения — это ассемблер с макрогенератором, система программирования I/O и программа распечатки содержимого памяти. Теперь рассмотрим те его элементы, которые используются при создании и выполнении программ — редактор и загрузчик. Завершая рассмотрение программного обеспечения, остановимся на средствах тестирования и отладки программ, потому что в них нуждается любая нетривиальная программа или система.

8.1. РЕДАКТОР

Редактор текстов представляет собой мощное средство для создания и модификации исходных текстов программ и других текстовых материалов. Используя вводимые с терминала команды, с помощью редактора можно записывать и удалять символы, строки или даже группы строк, содержащиеся в буфере. Во всех системах редактор работает в режиме «on-line», его ответ на команды является немедленным и динамичным.

Хороший редактор обеспечивает высокую эффективность труда программиста, его использование превращает такое устройство, как печатающий терминал, в удобное средство подготовки текстов. Редактор позволяет не только записывать и удалять символы и строки, но и осуществлять поиск нужных символов, выполнять исправления и читать или записывать блоки данных.

Большинство подобных программ имеют сходные характеристики, но мы, в частности, остановимся на редакторе операционной системы RT-11 и опишем некоторые его команды и их синтак-

сис. У редактора системы RT-11 имеется внутренний буфер, снабженный указателем текущего символа, т. е. последнего введенного или отредактированного пользователем

Одни редакторы работают со строками, другие — с отдельными символами. В первом случае необходим специальный символ конца строки, например возврат каретки. Такой редактор предполагает, что строка начинается с символа, следующего за возвратом каретки, и заканчивается символом, предшествующим возврату каретки.

Для редактора системы RT-11 типичны следующие команды:

1. INSERT — ввести новую строку символов. *Пример*: Itext$ вставляет «text» после текущего символа. Заметим, что каждая команда заканчивается нажатием клавиши ESC, в ответ на это редактор печатает знак «$».

2. DELETE — удалить строку символов. *Пример*: nD$ удаляет n следующих за указателем символов, если n положительное, или n предшествующих символов, если n отрицательное.

3. CHANGE — заменяет один набор символов другим. *Пример*: nCtext$ заменяет следующие за указателем n символов, если n больше нуля, или предшествующие n символов, если n меньше нуля, строкой «text».

4. KILL — уничтожает указанное число строк, начиная с позиции указателя. *Пример*: nK$ уничтожает n строк, следующих за указателем, если n положительное, и n предшествующих строк, если n отрицательное.

5. GET или FIND — обнаруживает первое или n-е появление строки. *Пример*: nGtext$ обнаруживает n-е появление строки «text».

6. LIST — печатает строку символов. *Пример*: nL$ печатает n строк, начиная с текущей позиции указателя.

7. VERIFY — распечатывает строку после изменения или обнаружения. *Пример*: V$.

8. READ — заполняет буфер редактора, читая блок данных с какого-либо внешнего устройства. *Пример*: ERfile$R$ инициализирует чтение первого блока из файла «file» с системного устройства.

9. WRITE — записывает содержимое буфера на внешнее устройство. *Пример:* EWfile$W$ инициализирует запись текущего блока текста в файл «file» на внешнее устройство.

10. EXIT — заканчивает редактирование текущего файла. *Пример*: EX$ заканчивает редактирование файла.

Кроме этих команд, существуют команды управления положением указателя, перечисленные ниже:

1. BEGIN — устанавливает указатель в начало буфера.

2. JUMP — перемещает указатель на заданное число символов в буфере.

3. ADVANCE — перемещает указатель на заданное число строк. Все команды должны заканчиваться двойным нажатием клавиши ESC. Это означает, что можно объединить несколько

команд в одной строке, а затем, вторым нажатием ESC, исполнить их.

Мощным средством редактора является возможность использования макрокоманд. Допустим, что пользователь хочет заменить в тексте *все* строки «Morse» строками «Morris». Для этого с помощью редактора RT-11 он должен, во-первых, очистить буфер макрокоманд командой M//$$, затем ввести макрокоманду M/FMorse$ — 5CMorris$/$$, перевести указатель в начало буфера B$$ и, наконец, выполнить команду 9999EM, что означает повторить макрокоманду 9999 раз. Редактор просмотрит весь файл потому, что действие команды «F», т. е. FIND, распространяется на весь файл, и во всех случаях, когда встретится строка «Morse», она будет заменена.

Степень развитости редактора в сильной мере определяется тем, в каких условиях он работает. Некоторые вычислительные машины допускают максимальную гибкость редактора, включая полную или частичную аббревиатуру команд, конкатенацию командных строк, работу с файлами и другие более тонкие методы редактирования. Есть вычислительные машины, имеющие очень сжатые однобуквенные команды, ограниченные размеры буфера, жесткий формат команд. Однако в любом случае редактор любой мини-ЭВМ обладает достаточной гибкостью для создания и модификации текстов программ.

8.1.1. Работа с редактором RT-11

Редактор RT-11 имеет много полезных атрибутов, в том числе макросредства, позволяющие заносить и удалять строку символов из внутреннего буфера. Редактор имеет командный режим, в котором он выполняет запросы пользователя (в этом режиме в начале каждой строки печатается символ «*»). На примере написания текста довольно тривиальной программы на рис. 8.1 продемонстрируем использование подмножества команд редактора. В примере предполагается, что редактор уже вызван и работает и что все непечатаемые символы (такие, как перевод строки, табуляция, возврат каретки) не отображаются. Кроме того, для удобства чтения справа приведен комментарий. Заметим снова, что каждая команда заканчивается символом «ESC», обозначаемым при печати знаком «$», а двойное нажатие клавиши «ESC» приводит к исполнению ранее набранной строки команд. Этот короткий пример, не будучи исчерпывающим, дает представление о возможностях редактора RT-11 (см. рис. 8.1).

8.1.2. Экранные редакторы

Возможности видеотерминалов и интерактивных дисплейных процессоров, обеспечивающих воспроизведение изображения на экране электронно-лучевой трубки, положили начало созданию ряда *экранных редакторов*. Ограничение традиционных редакторов,

ориентированных на применение печатающих терминалов, состоит в том, что, работая с ними, пользователь может в данный момент видеть и редактировать только одну строку. Однако в связи с тем, что программы имеют определенную структуру, выражающуюся в корреляции между строками и в некоторых других особенностях, целесообразно, чтобы пользователь видел бы сразу целый *блок* кодов. Значение этой возможности особенно возрастает

```
*I          .TITLE     START     ПОЛЬЗОВАТЕЛЬ ПЕРЕВОДИТ РЕДАКТОР
            R0 = %0              В РЕЖИМ ВВОДА И ПЕЧАТАЕТ
            R1 = %1              ВХОДНЫЕ СТРОКИ
            MOV        #1,R0
            CLR        R1
            CMP        R0,R1
            .END       START
$$                               ДВОЙНОЕ НАЖАТИЕ КЛАВИШИ «ESC»
                                 ЗАКАНЧИВАЕТ ВВОД, ПЕРЕВОДИТ РЕ-
                                 ДАКТОР В КОМАНДНЫЙ РЕЖИМ
*B$$                             ПЕРЕЙТИ В НАЧАЛО БУФЕРА
*3A$L$$
            MOV        #1,R0     СМЕСТИТЬСЯ НА ТРИ СТРОКИ И НАПЕ-
                                 ЧАТАТЬ ТЕКУЩУЮ СТРОКУ
*ISTART:$0A$L$$                  ДОБАВИТЬ МЕТКУ, ВОЗВРАТИТЬ КУРСОР
                                 В НАЧАЛО СТРОКИ И НАПЕЧАТАТЬ ЕЕ
START:  MOV        #1,R0
*GRO$—C0$0A$L$$                  НАЙТИ «RO», ЗАМЕНИТЬ НА «R0»,
                                 ПЕРЕМЕСТИТЬ КУРСОР В НАЧАЛО
                                 СТРОКИ И НАПЕЧАТАТЬ ЕЕ
START:  MOV        #1,R0
*EWDK1:FILE.MAC$EX$$             НАПИСАТЬ РЕЗУЛЬТАТ РЕДАКТИРОВА-
                                 НИЯ ПОД ИМЕНЕМ «FILE. MAC» НА DK1
.                                И ВОЗВРАТИТЬСЯ В МОНИТОР
```

Рис. 8.1. Пример работы с редактором RT-11

при редактировании тех программ, которые содержат ветвящиеся структуры. Экранные редакторы позволяют пользователю одновременно видеть блок кодов от 24 до 80 строк (окно). Курсор, контрастный по отношению к основному фону, ясно выделяет позицию, куда может быть введен новый символ, и такой символ сразу же однозначно отображается в поле окна.

Когда ЭВМ, работающая под управлением RT-11, имеет в своем составе графические процессоры VT-11 и VS-60, можно использовать специальные версии редакторов. В этом случае экран разделен на два поля, в одном отображаются 20 строк из текстового буфера, мерцающим курсором индицируется позиция указателя в нем, а второе служит для отображения введенных команд. При переходе в *текстовой режим* двойным нажатием клавиши ESC пользователь может легко перемещать курсор и вводить текст в занимаемую им позицию.

Если работа происходит под управлением систем RT-11 или RSX-11, то при наличии видеотерминалов VT-52 или VT-100 можно применить другой редактор TECO со специальными макросредствами, ориентированными на работу с VT-52 в экранном режиме. В некоторых отношениях у такого редактора имеются

преимущества по сравнению с редактором RT/VT. Он отображает на экран 24-строчную страницу текста и в отличие от редактора RT/VT использует для ввода команд специальную функциональную клавиатуру, имеющуюся у перечисленных терминалов. По отношению к основной клавиатуре редактор всегда находится в режиме готовности к приему текста, а по отношению к функциональной — к приему команд. Расположение и функции дополнительной клавиатуры показаны на рис. 8.2. Заметим, что ввод строк CTRL, BACKSP и DELETE здесь выполняется нажатием одной клавиши. Работа с этим редактором значительно сокращает время редактирования.

Функциональная клавиатура VT-52

«BLUE» Сохранить текст*	«RED» Команда TECO	«GREY» Вставить текст	«∧» Вверх по столбцу*
«7» Открыть новую строку*	«8» Новая страница*	«9» Сменить страницу*	«∨» Вниз по столбцу*
«4» Вверх на строку*	«5» Удалить символ*	«6» Удалить последнее	«>» Курсор вправо*
«1» Начало страницы*	«2» Конец страницы	«3» Начало строки	«<» Курсор влево»
«0» Вниз на строку*		«.» Поиск*	«ENTER» Аргумент поиска*

Другие клавиши

CTRL/C	Выход из макро.
CTRL/D	Уничтожить остаток строки*
CTRL/K	Уничтожить строку*
CTRL/U	Уничтожить начало строки
CTRL/Z	Выход из макро
BACKSP	Перейти в конец строки*
DELETE	Уничтожить предыдущий символ
	Двойное нажатие ESC повторяет команду TECO

Аргументы

Все помеченные звездочками команды могут иметь дополнительные аргументы в форме ⟨ESC⟩ ⟨цифры⟩

Рис. 8.2. Функциональная клавиатура экранного редактора TECO VT-52

Курсор, постоянно отображаемый на фоне 24-строчного окна и указывающий адресуемое место текстового буфера, может быть переведен в начало предыдущей строки (нажатием клавиши «4» или [4KP]), на следующую строку [0KP], в начало [1KP] или в конец [2KP] текущей страницы. Для перемещения курсора влево или вправо по текущей *строке* выполняются команды [<KP] и [>KP], а для перемещения его вверх или вниз по текущему *столбцу* — [∧KR] и [∨KP] соответственно. При использовании команды [8KP] текущая страница записывается в выходной файл, а в текстовый буфер заносится новая, которая отображается в

24-строчном окне. Если какой-либо команде предшествует нажатие клавиши ESC и ввод числа n, то команда выполняется n раз и каждый из них сопровождается выводом на экран символа «*».

Процедура поиска повторяющейся последовательности символов при редактировании предельно упрощена. Для этого нужно нажать клавишу «ENTER» и тогда в левом верхнем углу экрана в виде слова «Search:» (поиск) появится запрос на ввод искомой последовательности символов. В ответ на него пользователь должен набрать строку символов и нажать клавишу «.» на функциональной клавиатуре, что соответствует команде [.KP]. После этого и курсор *и* само окно будут перемещены таким образом, чтобы на экране оказалась отмеченной затребованная последовательность символов. Если повторять команды [.KP], то отмеченными окажутся следующие случаи повторения искомой последовательности, а если выполнить команду ESCn[.KP], то курсор сразу будет переведен на n-е повторение последовательности. При выполнении команды [6KP] стирается последняя найденная строка. Содержимое буфера поиска сохраняется до выполнения в него новой записи. Используя клавиши на функциональной клавиатуре, обозначенные как [BLUE] — голубая и [GREY] — серая, можно запоминать во внутреннем буфере n строк (ESCn[BLUE KP]) и затем вставлять их в нужное место текста (ESCn[GREY KP]). Содержимое внутреннего буфера сохраняется до тех пор, пока в него не будет сделана новая запись.

8.2. ЗАГРУЗЧИК

Проблемы начальной загрузки уже обсуждались в гл. 7 в связи с запуском системы. Хотя такой начальный загрузчик может быть также использован и для загрузки небольших программ, его универсальность и гибкость недостаточна при работе с большими программами. Для решения этой задачи служит *абсолютный загрузчик.*

Абсолютный загрузчик представляет собой системную программу, которая дает возможность программисту загружать его программы в любую доступную область памяти в произвольной последовательности. Он применяется для загрузки программ в двоичной форме с фиксированными адресами ячеек или в позиционно-независимом формате. После выполнения своей задачи процессор останавливается или управление передается во вновь загруженную программу.

Обычно абсолютный загрузчик загружается начальным загрузчиком в самую верхнюю область памяти и таким образом предотвращается пересечение по используемой памяти между ним и другими системными и пользовательскими программами. Это следует учитывать при написании пользовательских программ.

Для загрузки абсолютным загрузчиком любая программа должна состоять из нескольких блоков данных, каждый из которых включает:

1) индикатор начала блока;
2) число слов или байт, которые должны быть загружены;
3) адрес загрузки;
4) загружаемую информацию;
5) контрольную сумму.

В принципе можно обойтись без первого и последнего элемента, но, как правило, они являются необходимым требованием со стороны загрузчиков мини-ЭВМ.

Индикатор начала указывает на то, что за ним следует загружаемый блок; это дает возможность различать загружаемую и незагружаемую информацию. Например, если в качестве единственного устройства I/O в системе используется телетайп, то на выходной перфоленте может быть и листинг ассемблера и двоичный код программы, а разделение этой информации выполняется при загрузке.

Контрольная сумма может служить для обнаружения ошибок загрузки. При генерации абсолютным ассемблером двоичного кода такая сумма накапливается для каждой записи (обычно байта), а затем формируется контрольная сумма блока. При загрузке программы контрольная сумма заново вычисляется и сравнивается с записанной; при несовпадении фиксируется ошибка чтения. Если обнаружена ошибка, то дальнейшая загрузка прекращается. Таким образом, контрольная сумма служит гарантией правильности выполнения загрузочных операций.

Остальные элементы блока употребляются в соответствии с блок-схемой, показанной на рис. 8.3. Заметим, что последний адрес загрузки в некоторых случаях применяется как адрес передачи управления по окончании процесса загрузки. Это определяется тем, заканчивается ли программа оператором .END LABEL или просто «.END». Один из способов индикации этого различия, принятый в абсолютном загрузчике PDP-11, состоит в том, что адрес загрузки устанавливается четным или нечетным в зависимости от того, является ли он адресом передачи управления или нет.

Альтернативой выбору адреса загрузки из вводимого блока может быть задание этого адреса при помощи консольного переключателя. Это позволяет размещать PIC-программы в области памяти, отличные от указанных в загружаемом блоке, т. е. PIC-программы могут быть введены в произвольное место памяти путем формирования действительного адреса загрузки в виде суммы двух указанных выше адресов.

8.2.1. Перемещение программ

Перемещение PIC-программ может быть не только полезно, но и необходимо. Например, оно позволяет пользователю загрузить

программу распечатки памяти (программу дампа) так, чтобы занимаемая ею область не пересекалась с распечатываемым участком памяти. В более общем случае на основе PIC-программ можно создавать отдельные PIC-сегменты, а потом собирать из них в памяти одну большую программу.

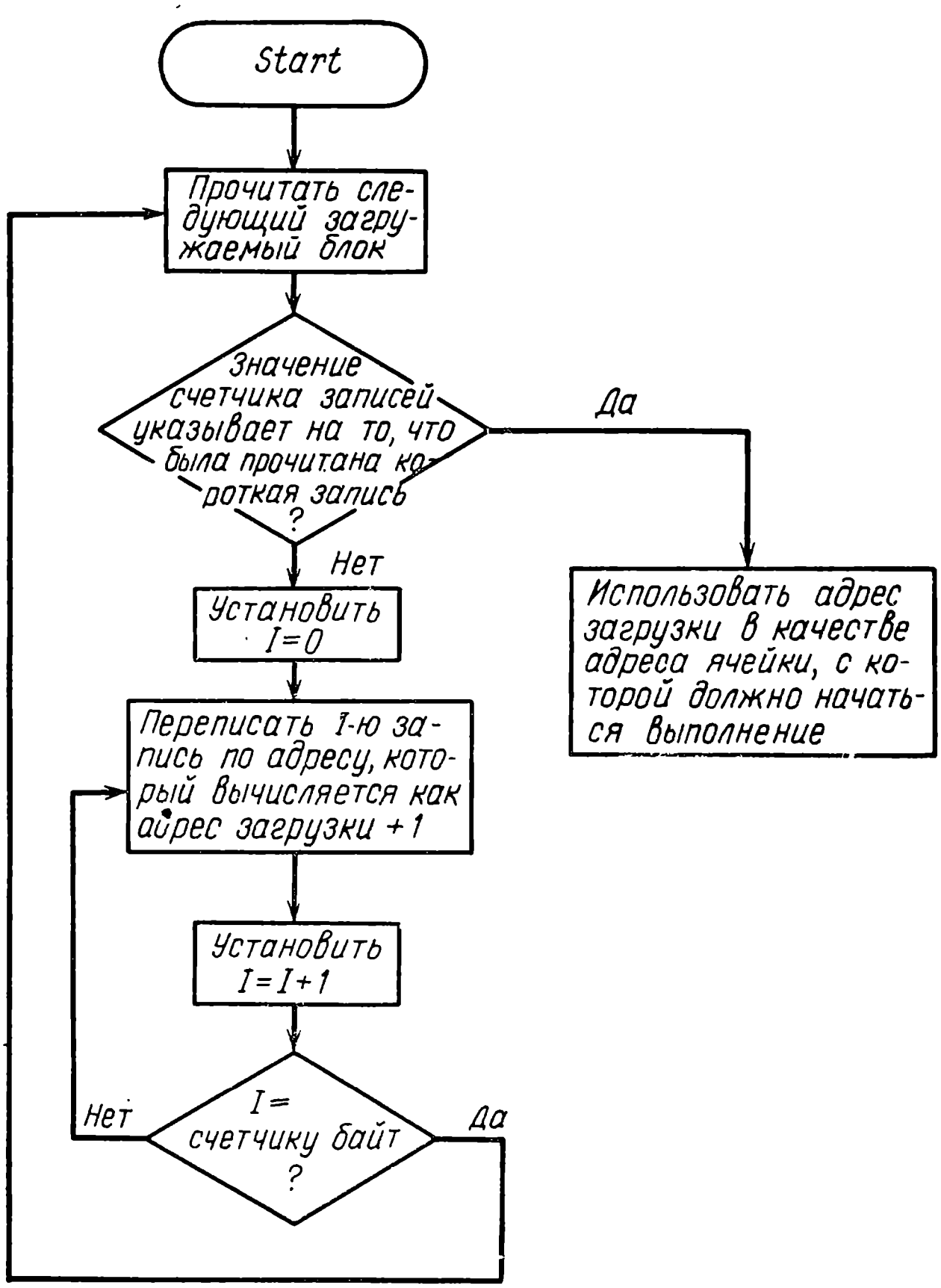


Рис. 8.3. Блок-схема простого загрузчика

Однако слишком жестоко заставлять программиста писать программы в PIC-формате. Было бы разумно оставить механический процесс превращения всех программ в перемещаемые за машиной, она вполне может решить эту проблему. Конечно же, программист должен так писать программы, чтобы связывающий загрузчик мог сделать их перемещаемыми.

Программисты, знакомые с Фортраном, знают, что каждая программа или подпрограмма, написанная на этом языке, должна компилироваться отдельно от других. Преимущества такого подхода состоят в следующем:

1. Если обнаружены ошибки в программе или подпрограмме, то только ее следует компилировать повторно.

2. Во время компиляции не требуется присвоения абсолютных адресов, что исключает возможность наложения программ по адресам. Это, в частности, позволяет менять размер подпрограмм, не оказывая при этом никакого влияния на связанные с ними программы.

3. Раздельная трансляция допускает включение в исходные тексты программ одних и тех же идентификаторов.

4. Однажды оттранслированная подпрограмма может быть помещена в библиотеку для общего пользования, и в последующем она не требует повторной трансляции.

Если в системе имеются *перемещающий ассемблер* (создающий перемещаемые программы) и *связывающий загрузчик,* то аналогичные процедуры могут быть проделаны и для программ, на языке Ассемблера *. Примером такой системы может служить RT-11.

До сих пор мы не рассматривали, каким образом реально загружаются подпрограммы и как выполняется редактирование связей с вызывающими программами. При абсолютном ассемблировании все программы и подпрограммы создавались совместно, для них выполнялось распределение памяти, а адреса подпрограмм хранились в таблице идентификаторов ассемблера. Если же каждая программа ассемблируется как отдельный модуль, то необходимо решить задачу распределения памяти для них и выполнить настройку адресов. Например, адресная часть команды

```
JSR REG,SUBR
```

должна быть настроена на реальный адрес входа в подпрограмму.

Перемещающий ассемблер и связывающий загрузчик решают эти задачи следующим образом. Во-первых, ассемблер производит объектный код в предположении, что начальный адрес каждой программы равен нулю. Во-вторых, он отмечает каждый относительный адрес или слово данных таким образом, что загрузчик будет знать, на какие части программы влияет перемещение. В-третьих, в ассемблере предусмотрена для программиста возможность объявления некоторых идентификаторов глобальными. Глобальный идентификатор объявляется либо в данной программе меткой или оператором прямого присваивания, либо это может быть сделано в другой программе, транслируемой отдельно. В первом случае глобальный символ называется *идентификатором входа*, а во втором — *внешним идентификатором.*

---

* Авторы рассматривают процедуры, выполняемые связывающим загрузчиком, в качестве основных для системы RT-11, тогда как она, аналогично другим современным операционным системам значительно больше ориентирована на использование редактора связи (программы LINK) и настраивающего загрузчика. Это же подтверждает и приведенный далее рис. 8.4. В целом материал разд. 8.2 более удачно рассматривается в книге: Д. Баррон. Ассемблеры и загрузчики. М., Мир, 1974. — *Примеч. ред.*

8.2.2. Редактирование связей и загрузка

В начале операции редактирования связей и загрузки используется следующая информация перемещающего ассемблера:

1. Объектный код программы.
2. Данные о свойствах перемещения отдельных полей в объектном коде.
3. Относительный адрес первой команды или элемента данных в загружаемом модуле.
4. Глобальные точки входа и ссылки на внешние символы.
5. Длина загружаемого модуля.

```
RT-11 LINK     V04-04A     LOAD MAP
NINEMA.SAV                 06-AUG-78

SECTION ADDR     SIZE     ENTRY    ADDR     ENTRY    ADDR     ENTRY    ADDR

  ABS. 000000   001000    $USRSW   000000   $RF1C1   000000   .V011S   000001
                          $NLCHN   000006   $HRDWR   000006   $WASIZ   000075
                          $LRECL   000210   $TRACE   004737

.$$$$. 001000   002170
XX     003170   000050
       003240   002514
       005754   013534   COEFF9   005754
       021510   001306   REALMA   021510
       023016   000016   ABS      023016
                   .
                   .
       024742   000132   SQRT     024742
                   .
                   .
       025074   000026   DIF$PS   025074   DIF$MS   025100   DIF$IS   025112
                         DIF$SS   025116   $DVR     025116
       032076   000340   EXP      032076
       032536   000354   COS      032536   SIN      032572
       033112   000102   AINT     033112   $INTR    033130
                   .
       033734   000036   DII$PS   033734   DII$MS   033742   DII$IS   033746

TRANSFER ADDRESS = 003240
HIGH LIMIT = 033772
```

Рис. 8.4. Пример листинга программы LINK RT-11

Эта информация необходима связывающему загрузчику для создания схемы загрузки, так как детализирует, какие программы должны быть загружены, каковы их длины, где они размещаются в памяти и какие другие программы требуются для их выполнения. Загрузчик работает до тех пор, пока не будут загружены все программы и не останется неудовлетворенных ссылок или же нужной ему программы не окажется среди доступных. Для облегчения процедур редактирования связей в ассемблере имеются две директивы «.GLOBL» и «.CSECT». Первая объявляет идентификаторы глобальными для всех загружаемых модулей, а вторая служит для описания отдельных характеристик загружаемого модуля и делает его имя глобально известным (в отличие от директивы «.ASECT», которая служит для объявления абсолютной секции, а все входящие в нее адреса задает как фиксированные).

Некоторые программы могут быть частью системной или пользовательской библиотеки. В состав таких библиотек входят оттранслированные программы пользователя и такие функции, как SIN, COS, TAN, EXP, LOG и т. д. Для организации доступа загрузчика к этим программам необходим *справочник*, содержащий адреса, точки входа и размеры программ, на которые могут быть сделаны ссылки. Некоторые из библиотечных программ, например SIN и COS, могут иметь общие коды. Типичные схемы распределения памяти и справочник показаны на рис. 8.4.

Заметим, что секции, имеющие имена, присвоенные посредством .CSECT, приведены в соответствующих им модулях с указанием базового адреса загрузки, символических точек входа, размера и действительного адреса в памяти. Все идентификаторы, объявленные в модуле как глобальные, описываются и в символических, и в действительных адресах. Две последние строки служат для указания начального адреса загружаемого модуля и адреса последней занимаемой им ячейки.

Реализовать загрузку и редактирование адресов можно двумя методами. Первый называется *методом векторов передачи,* он использует приемы, подобные таблице переходов, показанной в гл. 4. Помещая вызов каждой внешней программы в таблицу переходов, загрузчик в конечном итоге может сформировать таблицу, содержащую адреса всех загруженных программ. На рис. 8.5 показан отрывок программы на ассемблере PDP-11, который может служить для получения перемещаемого кода, содержащего таблицу перехода к двум программам: MULT и DIVD. После загрузки строки этой таблицы будут содержать действительные адреса переходов к начальным ячейкам программ MULT и DIVD.

Код на ассемблере		*Результат в перемещаемой форме*			*Программа в памяти (адрес MULT-200, DIVD-300)*		
		0 MULT: 0			100	.WORD	200
.GLOBL	MULT,DIVD	2 DIVD: 0			102	.WORD	300
ADD	A,B	4	ADD	50,52	104	ADD	150,152
JSR	PC,@MULT	12	JSR	PC,@0	112	JSR	PC,@100
SUB	B,C	16	SUB	52,54	116	SUB	152,154
JSR	PC,@DIVD	24	JSR	PC,@2	124	JSR	PC,@102
JSR	PC,@MULT	30	JSR	PC;@0	130	JSR	PC,@100
•			•			•	
•			•			•	

Рис. 8.5. Загрузка с использованием векторов передачи

Второй метод используется в связывающих загрузчиках и основывается на стремлении избежать одного уровня косвенной адресации, необходимого для векторов передачи. С этой целью строится связанный список вызовов внешних программ, он сохраняется до тех пор, пока относительные адреса внешних программ не станут известны. Теперь загрузчик разворачивает список и формирует прямые вызовы внешних программ.

На рис. 8.6 показан тот же пример для PDP-11, что и на рис. 8.5, но на базе другого приема загрузки. Перемещаемый объектный код включает список ссылок к одной и той же внешней программе, который заканчивается нулевым элементом, обозначенным на рисунке «—». Принципиальная разница между рассмотренными методами состоит в том, что при использовании первого из них разрешение внешних связей выполняется во время загрузки, а второго — до нее; причем результатом работы редактирующей части является законченный загрузочный модуль, который читается в память собственно загрузчиком.

Код на ассемблере		*Результат в перемещаемой форме*		*Программа в памяти* (адрес MULT-200, DIVD-300)			
		MULT	L24				
.GLOBL	MULT,DIVD	DIVD	L20				
ADD	A,B	0	ADD	50,52	100	ADD	150,152
JSR	PC,MULT	6	JSR	PC,-	106	JSR	PC,200
SUB	B,C	12	SUB	52,54	112	SUB	152,154
JSR	PC,DIVD	20	JSR	PC,-	120	JSR	PC,300
JSR	PC,MULT	24	JSR	PC,L6	124	JSR	PC,200

Рис. 8.6. Загрузка с помощью связующего загрузчика

В любом случае получаются одни и те же результаты:

1) объектные модули настраиваются на абсолютные адреса;
2) различные модули объединяются вместе и устанавливается взаимосвязь по глобальным идентификаторам между теми модулями, в которых они определены, и теми, которые их используют;
3) формируется карта распределения памяти, иллюстрирующая присвоение абсолютных адресов.

Все это позволяет программисту ассемблировать программы и подпрограммы как отдельные модули, но выполнять их — как одну программу.

8.3. МЕТОДЫ ОТЛАДКИ

Ни одна программа, какой бы сложной она ни была, не будет работать верно при первом прогоне — это одно из основных правил программирования. Дело в том, что даже синтаксически корректная программа может содержать логические ошибки, не позволяющие ей выполнять предписанные действия. Обнаруживая синтаксические ошибки, транслятор не умеет находить логических. Они могут быть выявлены только программистом и только при выполнении программ на машине.

Не всегда просто выяснить, содержит ли программа логические ошибки. В основном ЭВМ используются для решения проблем, требующих вычислений, и для проверки программ нужно хорошо знать, какие результаты должны быть получены. Ошибка легко находится только в том случае, если полученный

результат сильно расходится с предполагаемым. Если же несоответствие мало, то программист оказывается перед трудной задачей определения правильности работы программы. При проверке большой и сложной программы он не всегда в состоянии проконтролировать все возможные варианты и тогда ему не остается ничего другого, как принять на веру, что его программа работает до тех пор, пока не будет доказано обратное.

Если известно, что логическая ошибка существует, то возникает проблема определения ее причины. Для этой цели существует несколько методов:

1) распечатать содержимое ячеек памяти, относящихся к получению результатов;

2) используя переключатели и индикацию консоли ЭВМ, попытаться следить за выполнением программы;

3) проследить последовательность выполнения отдельных команд программы (выполнить ее трассировку);

4) ввести дополнительный вывод промежуточных результатов. Бывает полезна и распечатка памяти, но она статична и отражает уже случившиеся события. К моменту получения распечатки ошибка может привести к тому, что любая относящаяся к ней информация, в том числе и ее причина, может быть уничтожена или искажена.

При монопольном использовании машины допустима организация покомандного выполнения программы с проверкой содержимого любой ячейки памяти. Однако такая техника отладки занимает очень много времени и чрезвычайно утомительна, поскольку приходится считывать показания в двоичной форме и соотносить их с символическими выражениями в программе. Этот прием может быть несколько усовершенствован, если заранее определить точки остановки в программе, предшествующие проверяемой секции; тогда можно значительно уменьшить необходимое число операций. Конечно же, здесь от программиста требуется знание точек остановки.

Еще лучше организовать такой режим работы программы, когда распечатываются выполняемые команды и промежуточные результаты, получаемые при выполнении определенной траектории программы (трассировка или отслеживание). Это тоже довольно неэкономный по времени прием, однако только в смысле машинного времени, так как трассировка может выполняться в отсутствие программиста. В некоторых ЭВМ, например в PDP-11, в слове состояния имеется специальный Т-бит, служащий для обеспечения трассировки программы. Если он установлен в единичное состояние, то прерывание возникает после выполнения каждой команды программы, что значительно упрощает организацию трассировки.

Если машинное время ограничено, то при отслеживании должно быть выбрано ограниченное количество ячеек или переменных. В таких обстоятельствах либо используется специальная программа трассировки, либо программист сам вставляет в

программу операторы, сообщающие о том, что такие-то переменные изменили значение, к такой-то ячейке был выполнен переход или на нее была ссылка.

Разумеется, программист может, сидя за столом, по листингу программы попытаться ее мысленно выполнить. Этот прием часто применяется, но он эффективен только для небольших программ. Возможности человеческой памяти несравнимы с возможностями ЭВМ, и человеку трудно запомнить все шаги и все команды даже очень коротенькой программы.

Для отладки пользовательских программ необходима служебная программа, обеспечивающая удобное и точное выполнение тех же самых операций, которые пользователь мог бы делать с помощью консольных переключателей, при распечатке памяти и селективной трассировке программы. Такое средство называется *динамической отлаживающей программой* DDP.

На мини-ЭВМ DDP реализуется в виде диалоговой системной программы, предоставляющей удобные средства отладки и управления ходом выполнения программы. По существу, DDP представляет собой двоичный редактор и супервизор одновременно.

Посредством команд с терминала, адресованных DDP, пользователь может: 1) запустить программу, 2) прервать ее выполнение в заранее определенной точке, 3) проанализировать и изменить содержимое ячеек памяти и регистров, 4) сделать дополнения или коррекции в работающей программе, используя восьмеричный или символический код. Команды имеют следующую форму:

1. OPEN — анализирует или модифицирует содержимое ячейки памяти.
2. CLOSE — переход к следующей ячейке или другой операции DDP.
3. MODE — устанавливает тип входного или выходного режима операций.
4. BREAKPOINT — прерывает выполнение программы в заранее определенной точке.
5. SEARCH — выполняет поиск определенного набора бит, например адреса, константы, команды.
6. LIMIT — устанавливает пределы поиска (адреса в памяти).
7. BEGIN — начинает выполнение программы с определенной ячейки.
8. PROCEED — продолжает выполнение программы с точки, где было выполнено прерывание.

8.3.1. Пример использования отладочных средств

В системе PDP-11 используется отладочная программа ODT-11, типичная для программ такого рода, работающих на мини-ЭВМ. Как и редактор, ODT-11 имеет командный режим, в котором выполняются запросы к пользователю, обозначаемые на терминале звездочкой.

Основной набор команд ODT:

1) n/ — открывает слово;

2) cr — возврат каретки обозначает закрытие открытой ранее ячейки;

3) n; G — начинает выполнение с ячейки n;

4) n; B — устанавливает точку прерывания в ячейке n;

5) ;P — продолжает выполнение с точки прерывания;

6) $n/ — открывает регистр n.

Рассмотрим работу ODT-11 на примере следующей простейшей программы на языке Ассемблера:

```
        .=1000
START:  MOV     #1,R0
        CLR     R1
        CMP     R0,R1
        HALT
        .END
```

В программе директива .END не имеет метки, поэтому ODT-11 начнет выполнение от имени этой программы. В процессе работы возможен, в частности, следующий диалог (комментарии добавлены для большей наглядности):

```
*1004/005001          ПРОВЕРЯЕТ КОМАНДУ
*$1/000000 123456     МЕНЯЕТ СОДЕРЖИМОЕ R1
*1004;B               УСТАНАВЛИВАЕТ ТОЧКИ ПРЕРЫВАНИЯ
*1010;B               В ЯЧЕЙКАХ 1004 и 1010
*1000;G               НАЧИНАЕТ ВЫПОЛНЕНИЕ С НАЧАЛА
B0,001004             ВЫПОЛНЯЕТСЯ ПРЕРЫВАНИЕ
*$0/000001            ПРОВЕРЯЕТ R0
*$1/123456            и R1
*P                    ПРОДОЛЖАЕТ ВЫПОЛНЕНИЕ ПРОГРАММЫ
B1;001010             СЛЕДУЮЩАЯ ТОЧКА ПРЕРЫВАНИЯ
*$1/000000            ПРОВЕРЯЕТ R1
*$0/000001            И R0
```

Хотя этот пример очень краток, он все же дает читателю представление о том, что такое динамическая отлаживающая программа и как она работает. При работе на мини-ЭВМ, где обычно имеется весьма ограниченное число индикационных лампочек на консоли и аппаратные средства анализа состояния памяти и регистров тоже весьма ограничены, программист быстро оценивает преимущества использования DDP по сравнению с работой с консоли.

8.4. ПРИКЛАДНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ И ЕГО ВЫЧИСЛИТЕЛЬНАЯ СРЕДА

Теперь, после того как были рассмотрены вопросы организации и программирования, наше внимание будет привлечено к решению при помощи ЭВМ нетривиальных задач. Проанализируем также ту вычислительную среду, в которой создаются, отлаживаются и выполняются программы, реализующие такие решения.

В этом же смысле нас будут интересовать редактирование, компиляция или ассемблирование программ, их компоновка и загрузка.

В гл. 9 рассматриваются вопросы использования ЭВМ в режиме реального времени. Здесь прикладные программы также служат для иллюстрации и закрепления представлений о таких идеях, как вычисления с управлением по прерываниям, прямой доступ в память, микропрограммирование. Кроме того, в этих разделах рассматриваются с точки зрения и программной и аппаратной периферийные устройства, часто применяемые в системах реального времени: аналого-цифровые преобразователи и графические процессоры.

Две последние главы посвящены серьезному анализу структур *операционных систем и систем мультипрограммирования и разделения времени.*

УПРАЖНЕНИЯ

1. В чем состоит различие между редакторами, используемыми для создания программ, и редакторами для подготовки текстового материала? Какие типы команд характерны для тех и других?

2. Используя в качестве примера редактор RT-11, перечислите его положительные и отрицательные качества. Дайте предложения по его усовершенствованию.

3. Напишите программу, реализующую функции абсолютного загрузчика в соответствии с блок-схемой на рис. 8.3.

4. Какие средства ODT-11 не обсуждаются в этой главе? Опишите методы их использования.

БИБЛИОГРАФИЯ

Одно из лучших исследований по редакторам текстов содержит статья Van Dam A., Rice D. E. (1971). Конечно, при использовании определенного редактора следует обращаться к соответствующим руководствам, например «PDP-11 Edit-11 Text Editor». Все операционные системы PDP-11 содержат детальные руководства по различным системным программам.

Глава 9

ИНТЕРАКТИВНЫЕ ВЫЧИСЛЕНИЯ В РЕАЛЬНОМ ВРЕМЕНИ

Несмотря на то, что ЭВМ проникли уже почти во все сферы человеческой деятельности, они по-прежнему «выдают» результаты при помощи кип бумаги, а основным средством общения с ними остается ввод данных с клавиатуры терминала. Однако в рассматриваемых в настоящей главе системах для *интерактивных вычислений реального времени* дело обстоит несколько иначе. Здесь *ввод* информации производится непосредственно в виде волновых колебаний (в том числе речи и других биологических сигналов) или с помощью косвенных источников, таких, как световое перо, джойстик или сколка. *Вывод* из ЭВМ осуществляется посредством аналоговых волновых колебаний (в частности речи), рисунков на электронно-лучевой грубке (CRT) или в некоторых случаях алфавитно-цифрового отображения на специальных дисплеях (плазменных панелях, светоиспускающих диодах и т. д.). В большинстве подобных систем осуществляются интерактивные взаимодействия оператора с ЭВМ в реальном масштабе времени. При этом широко используются прерывания и инициация и запуск задач на выполнение в зависимости от некоторых событий. Здесь также существенное значение имеет наилучшее использование особенностей архитектуры ЭВМ для достижения минимума времени выполнения программы. Так, от архитектуры ЭВМ зависит возможность решения задач в реальном времени, особенно в тех случаях, когда другие режимы неприемлемы.

Ниже рассматриваются особенности некоторых устройств ввода-вывода, используемых для взаимодействия с ЭВМ в реальном времени.

9.1. ИНТЕРАКТИВНЫЕ ГРАФИЧЕСКИЕ ПРОЦЕССОРЫ

DEC VT-11 — это автономный *дисплейный процессор* (DPU), который, подобно CPU PDP-11, имеет доступ к памяти и может выбирать на выполнение находящиеся в ней команды. DPU подключен к UNIBUS аналогично другим периферийным устройствам. В результате выполнения им команд из памяти получаются картины, отображаемые на CRT. Они либо представляют конечный результат некоторых вычислений, либо, что более важно, являются средством, помогающим оператору работать с другими

задачами в интерактивном режиме. В этом смысле интересно не то, что DPU выполняет некоторые действия, а то, как он это делает.

Например, VT-11 использует прямой доступ к памяти (DMA). Кроме того, в этом устройстве в значительной степени реализованы концепции микропрограммирования, обсуждавшиеся в предыдущих главах. И наконец, в нем существует возможность вводить информацию с помощью светового пера. Такие данные обрабатываются специальными подпрограммами при тесном взаимодействии CPU и DPU. На примере операций, выполняемых VT-11 DPU, можно проследить реализацию элементарных идей, присущих микропроцессорным системам вообще. А так как VT-11 является представителем дисплейных процессоров в целом, то полезно обсудить его множество команд и выполняемых операций. Заметим, что набор команд VT-11 является подмножеством команд дисплейного процессора VS-60 DPU и, следовательно, все программы VT-11 могут выполняться на VS-60.

9.1.1. Дисплейный процессор с обновлением изображения

Графический дисплей похож на обычный телевизор. Изображение на его экране повторяется 30 раз в секунду, т. е. с минимальной частотой, обеспечивающей отсутствие мерцания. С другой стороны, такая частота смены изображения обеспечивает возможность оживления картины, каждый раз слегка изменяя ее контур. В противоположность телевизионному изображению, состоящему из 525 горизонтальных линий различной интенсивности на разных участках линий, изображение DPU образуется произвольным числом линейных сегментов и других объектов, в том числе символов. Сегменты имеют различную ориентацию и интенсивность. К этому следует добавить, что луч CRT сканирует каждую точку изображения, облегчая таким образом взаимодействие пользователя с изображением.

DPU рисует картину, многократно сканируя *дисплейный файл,* состоящий из *команд*, или *инструкций,* и *данных.* Этот файл должен находиться в памяти подобно другим командам и данным PDP-11. Он загружается с внешнего устройства, как обычная программа, и, будучи загруженным, может изменяться в результате действий, выполняемых CPU.

Команды дисплейного файла могут вызывать следующие действия:

1) отклонять луч в любую точку 1024 × 1024-точечной координатной системы;
2) проводить линию от данной точки в соответствии со смещением и знаком x и y;
3) отображать группы и строки символов в коде ASCII на участке, определенном командами, находящимися в памяти типа ROM;

4) строить графики функций $f(x)$ или $f(y)$ в зависимости от увеличивающихся значений их аргументов.

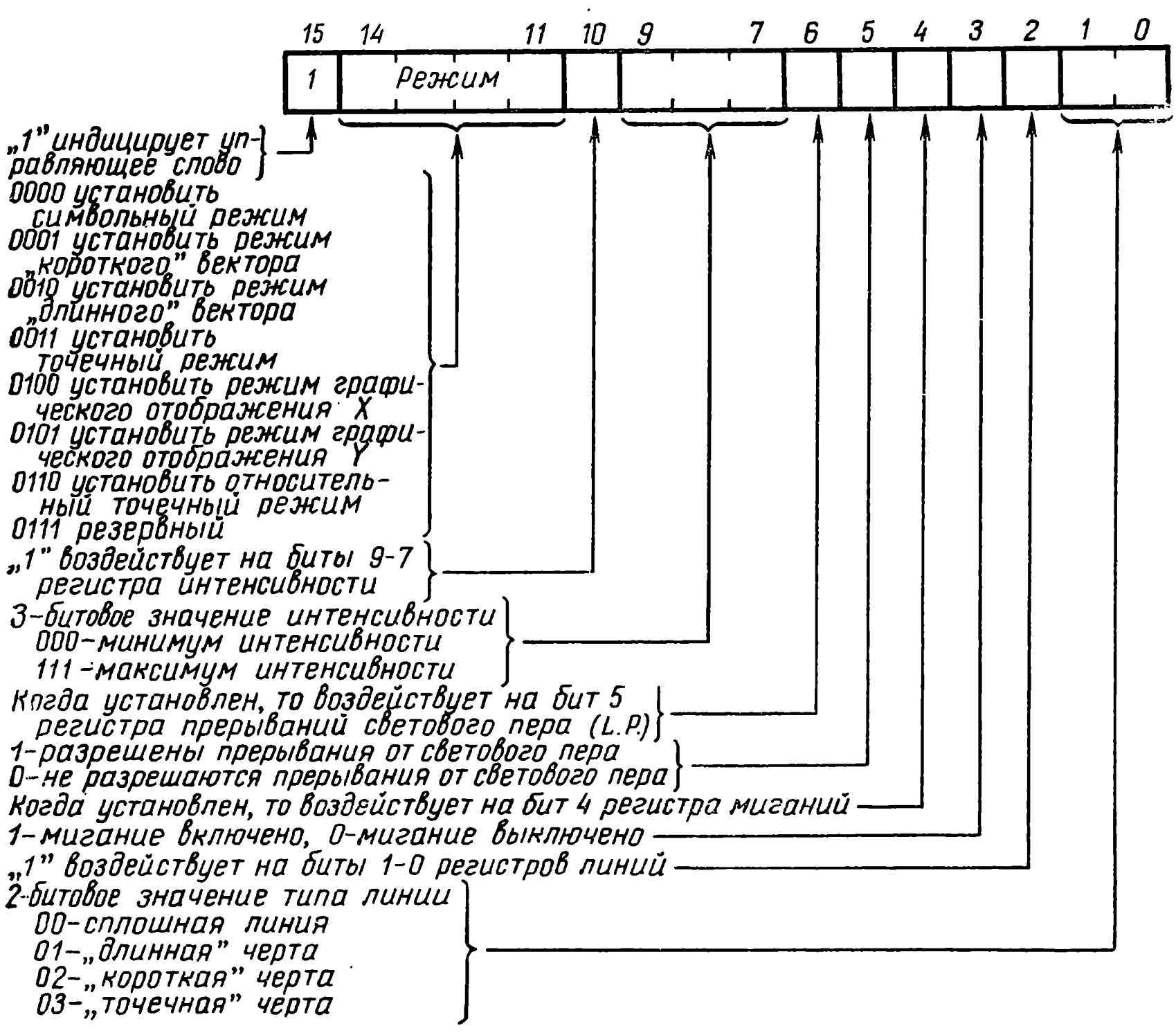
Для каждого рисунка дисплей обеспечивает от 1 до 8 степеней интенсивности и 4 вида линий (сплошная, состоящая из точек, пунктирная, штрихпунктирная). Кроме того, есть возможность изменять отдельные элементы изображения путем воздействия на CPU.

Для более глубокого понимания функционирования VT-11, проведем теперь анализ его инструкций и форматов данных.

9.1.2. Инструкции и форматы данных графического процессора VT-11

У всех *инструкций* DPU VT-11 MSB установлен в «1»:

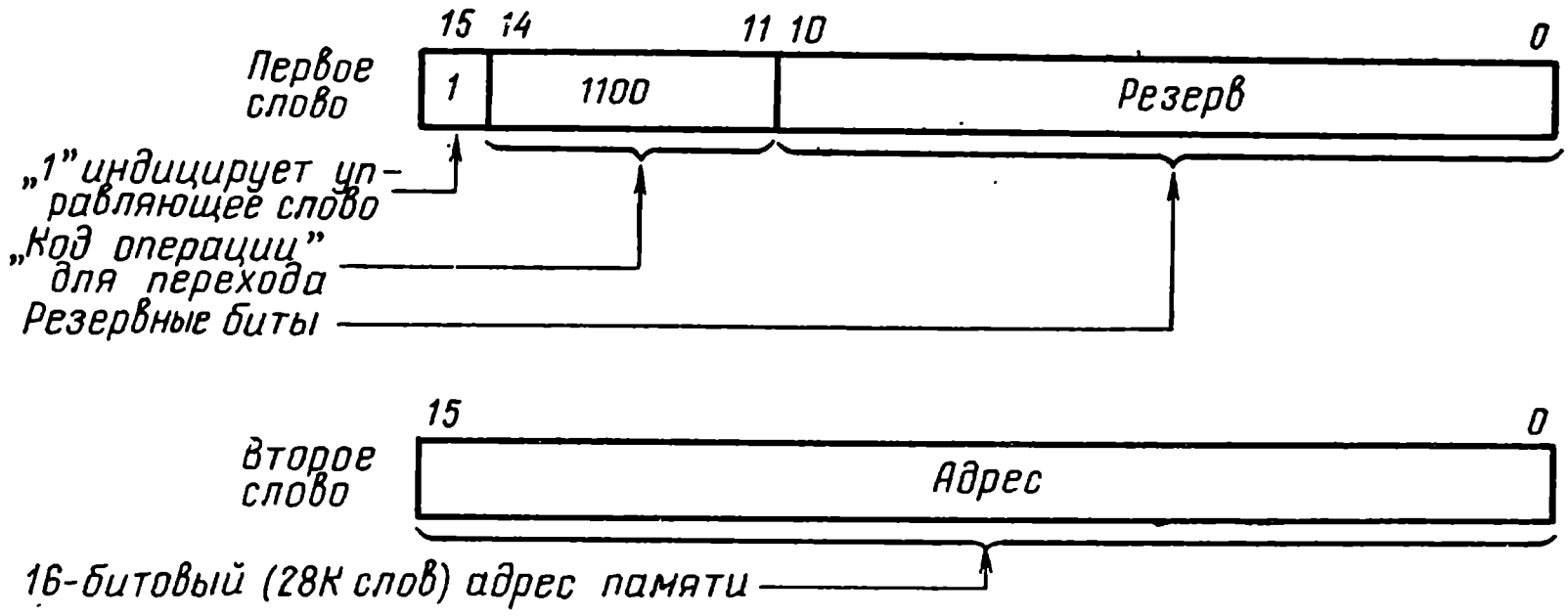
Инструкция SET GRAPHICS MODE (УСТАНОВИТЬ ГРАФИЧЕСКИЙ РЕЖИМ)



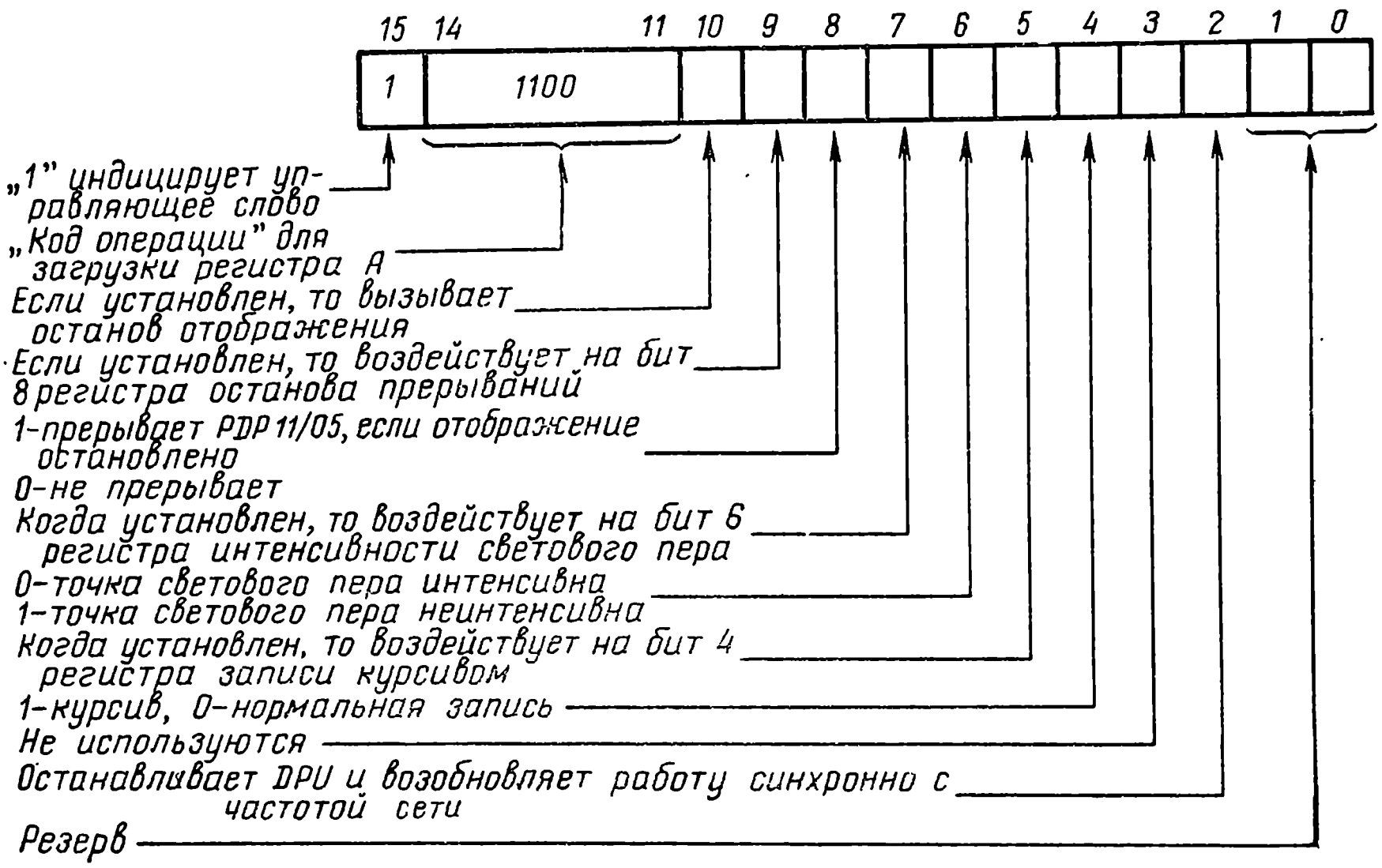
1) *определяет*, как дисплей должен интерпретировать 16-битовые слова данных, и/или

2) *устанавливает* биты в регистрах, воздействующих на атрибуты выводимого изображения (например, степень яркости, мигание, тип линии).

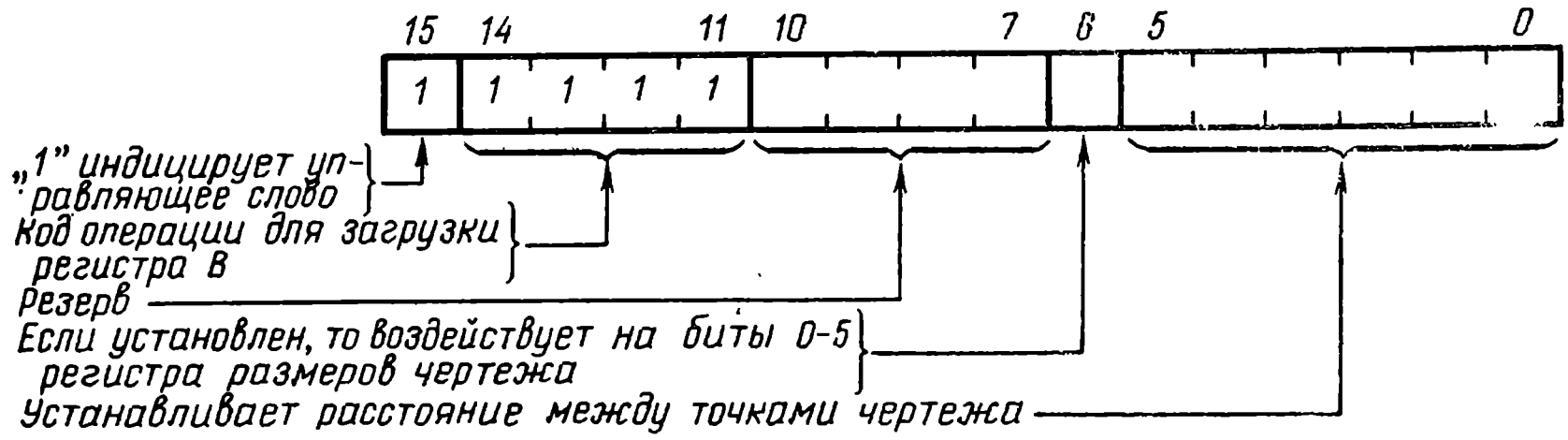
Инструкция DJMP вызывает загрузку регистра команд графического дисплея (DPC) адресом, непосредственно следующим за инструкцией DJMP. Обычно же в этот регистр загружается адрес очередной инструкции при переходе от одной ячейки памяти к другой.



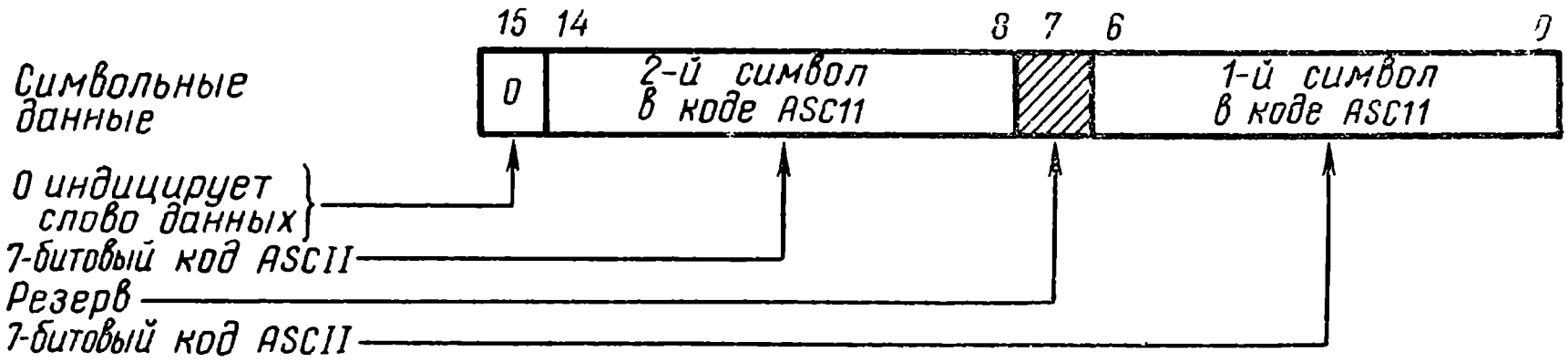
Инструкция LOAD STATUS REGISTER A (ЗАГРУЗИТЬ РЕГИСТР СОСТОЯНИЯ А) загружает еще один регистр, воздействующий на атрибуты отображаемых объектов или изменяющий действия DPU в зависимости от определенных условий.



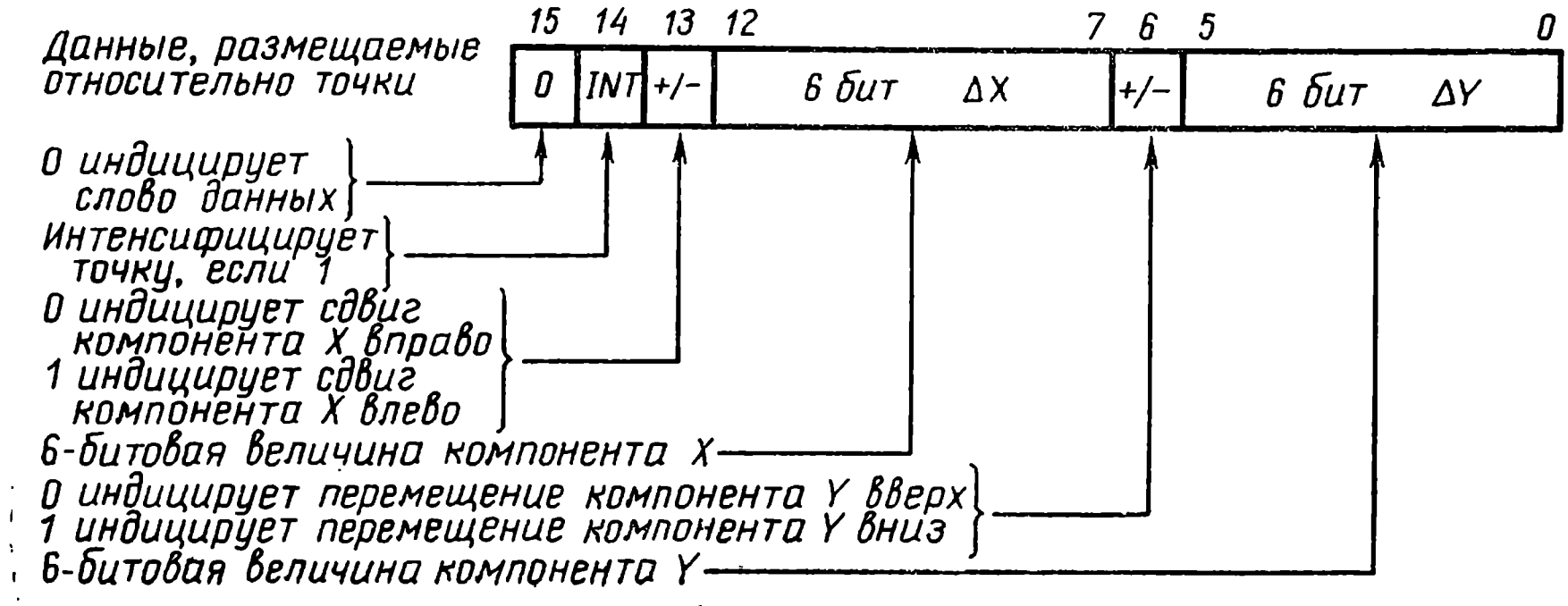
Инструкция LOAD STATUS REGISTER B (ЗАГРУЗИТЬ РЕГИСТР СОСТОЯНИЯ В) используется, когда надо форматировать вывод данных в графическом режиме.



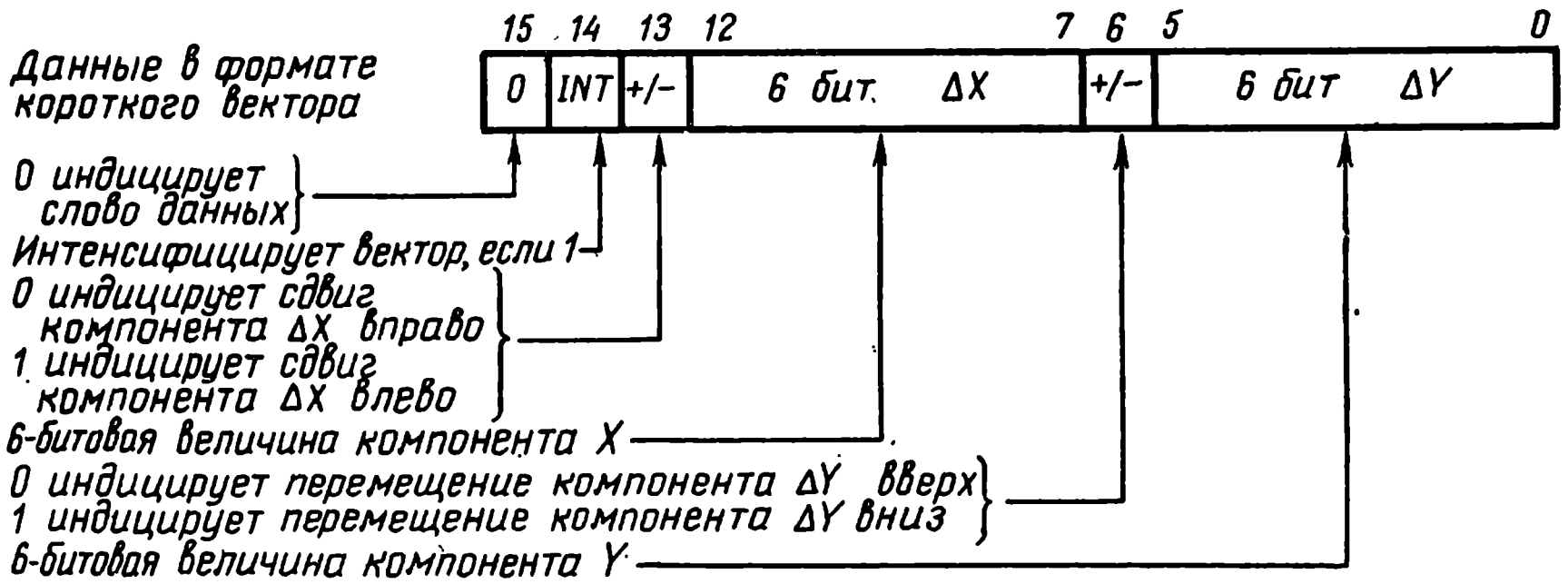
VT-11 в своей работе использует понятие «остаточный». Это значит, что текущим режимом будет режим, установленный последней встретившейся инструкцией «УСТАНОВИТЬ ГРАФИЧЕСКИЙ РЕЖИМ». Все слова *данных*, встретившиеся после нее, будут интерпретироваться в соответствии с текущим режимом. Любые данные в VT-11 имеют значение MSB, равное «0»:



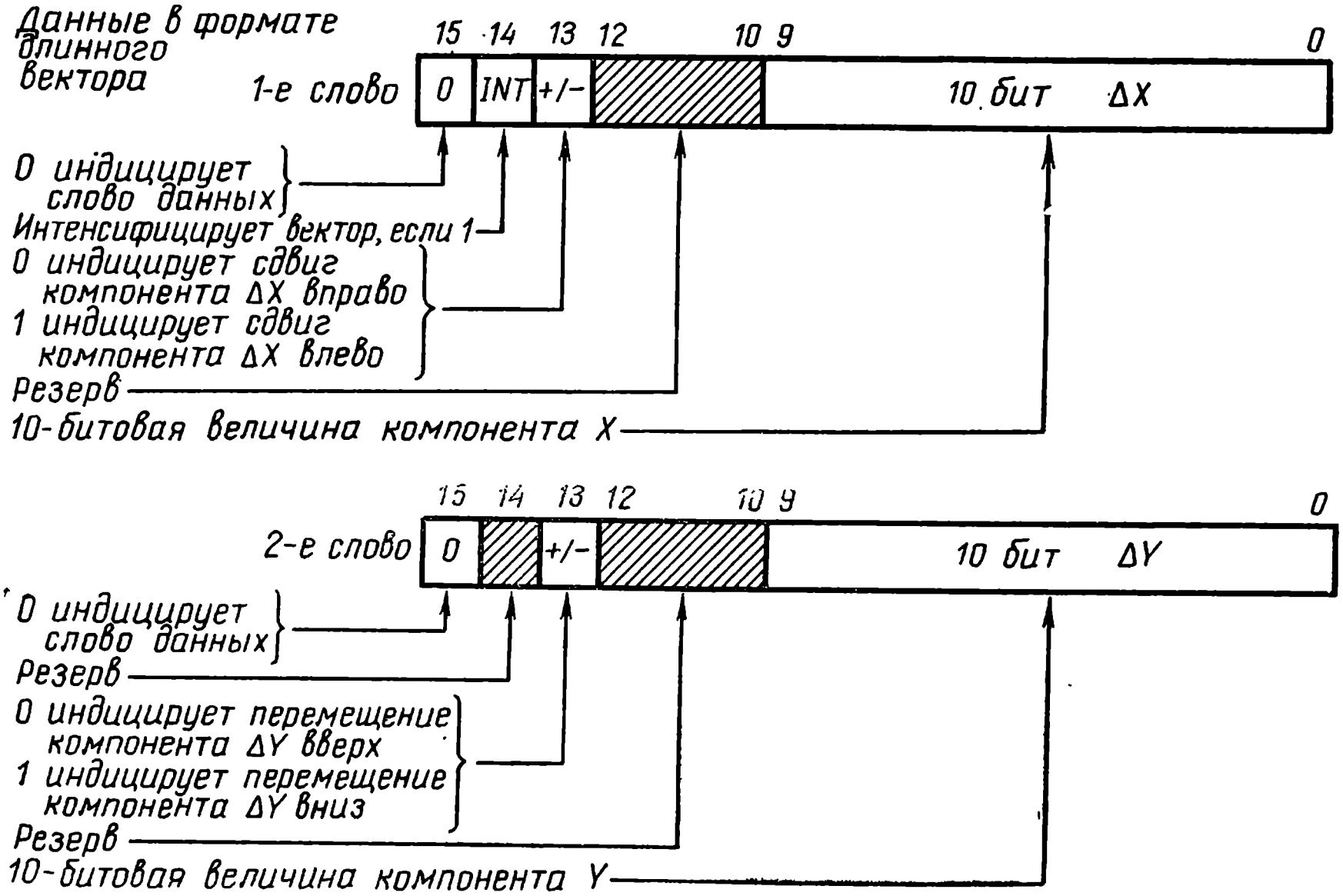
Каждое слово данных, которое встречает DPU, будет интерпретироваться в соответствии со сказанным выше и обеспечит изображение двух символов. Первым будет изображаться символ 1, начиная с того места, где находится текущая точка луча трубки. Это будет нижний левый угол экрана. Затем, справа от первого, будет отображен второй символ, и луч будет находиться в положении, определенном изображением второго символа.



Здесь 6 битов специфицируют величину смещения луча по осям *x* и *y* от текущей позиции. Бит 13 специфицирует направление *x* (0-вправо,) а бит 6 — направление развертки *y* (0-вверх). Таким образом, пока дисплей находится в режиме графического вывода, каждое слово данных будет интерпретироваться в соответствии со сказанным выше и изображается ровный видимый (бит 14 = 1) или невидимый вектор. Невидимые векторы позволяют производить относительные смещения луча.

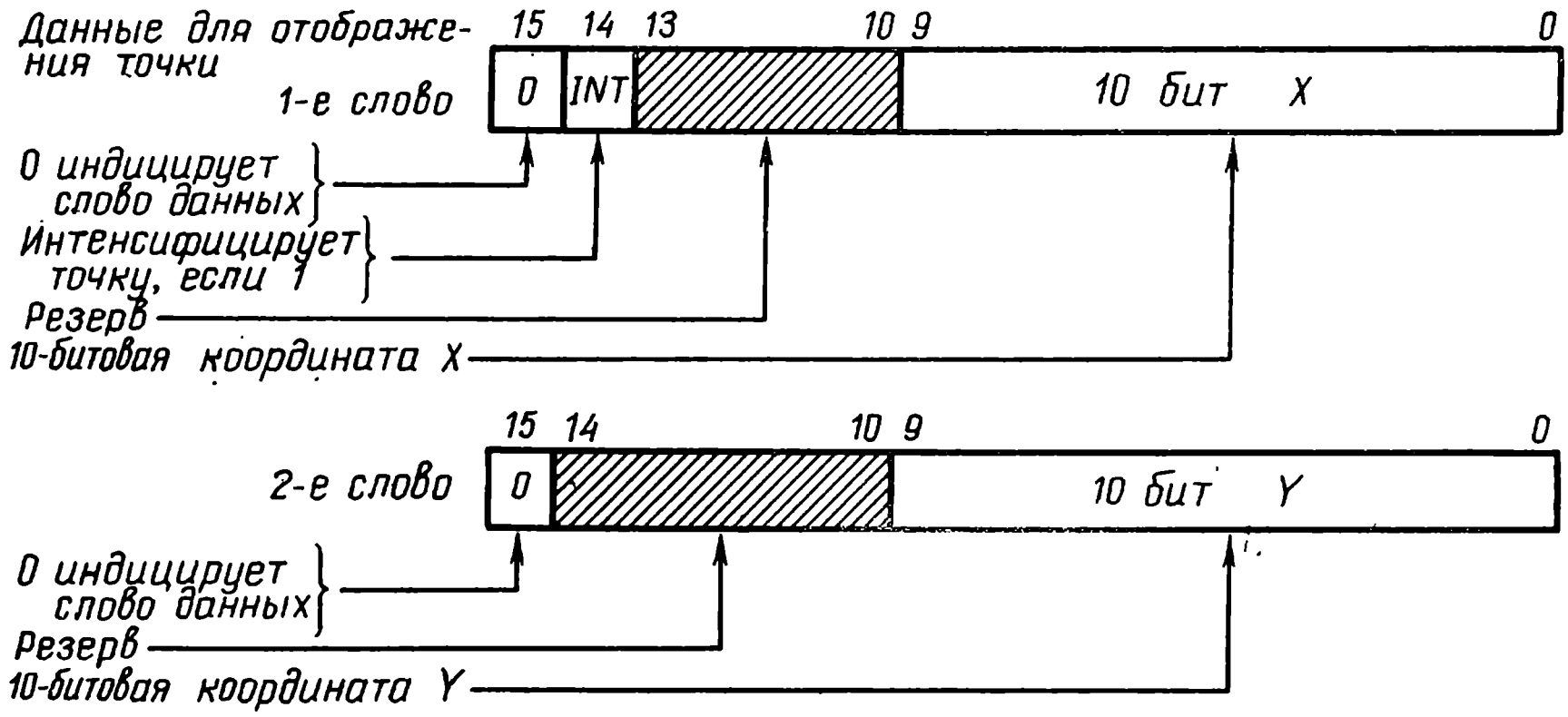


Луч смещается так же, как при выполнении инструкции SHORT VECTOR MODE, за исключением того, что отсутствует видимая линия. Если бит 14 = 1, то только видимая *точка* ставится в конечной позиции луча.

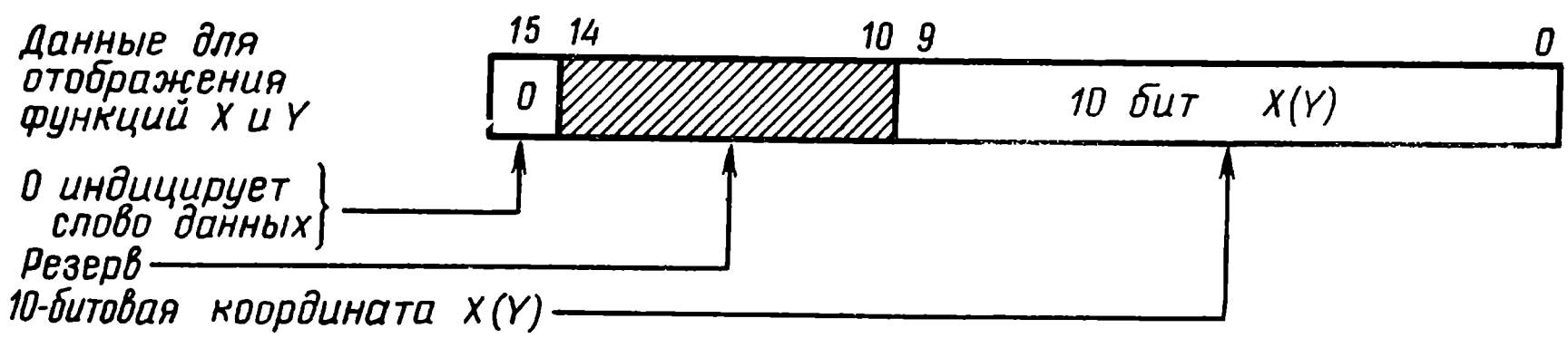


В режиме 2 каждая пара слов данных интерпретируется как длинный относительный вектор. Назначение битов направления и

величины то же самое, что и в инструкции SHORT VECTOR DATA. XXX..X означает неиспользуемые биты.



В режиме 3 каждая пара встретившихся слов данных интерпретируется так, что обеспечивает невидимое перемещение луча в позицию с координатами x и y. Если бит INT установлен в «1», то в конечной позиции луча ставится *точка*.



В режимах 4 или 5 каждое встретившееся слово интерпретируется как $f(x)$ или $f(y)$ соответственно и отмечается точкой. Значение координат x и y возрастает на величину, содержащуюся в регистре интервала между элементами изображения (GRAPHPLOT INCREMENT). Этот регистр в свою очередь загружается по инструкции LOAD STATUS REGISTER B. Таким образом, регистр интервала между элементами обеспечивает вывод данных в определенном формате.

В заключение можно отметить, что инструкции SET GRAPHIC MODE должны выдаваться в том случае, когда требуется отобразить новую картину. При этом реализуется «остаточный» принцип. Если выполняется видимое изображение, то анализируются все регистры, специфицирующие интенсивность, мигание, использование светового пера и типа линии. Полученные в итоге суммарные параметры учитываются при выводе. Это значит, что выдача команды на изменение изображения будет осуществляться в том случае, если какой-то из параметров изменится по сравнению с его настоящим значением. Наконец, отметим, что для упрощения

изложения все координаты указываются в десятичной системе счисления.

Нижеследующая короткая программа рисует в левом нижнем углу (с координатами 320, 320) квадрат 128 на 128 растровых единиц. Квадрат выполняется на 4-м уровне яркости, сплошными линиями, не мерцает и не предназначается для взаимодействия со световым пером. Мнемоника, необходимая для установления графических режимов, загрузки регистров состояния и определения данных, обеспечивается макросом «.GTDEF» системы RT-11. Дисплейный файл формируется программистом и загружается в память вместе с программой. Для начала работы DPU необходимо лишь, чтобы CPU с помощью команды MOV поместил начальный адрес дисплейного файла в счетчик команд дисплея (DPC) по адресу 172000. После этого DPU, многократно сканируя, «исполняет» файл, отображая результирующую картину. *CPU* в этом случае *совершенно свободен и может использоваться другими задачами.*

```
        START:  MOV     #A,@#DPC        ;LOAD DPC WITH STARTING ADDRESS
                                        ;OF DISPLAY FILE
        WAIT:   WAIT                    ;CPU NO LONGER NEEDED
                BR      WAIT            ;BACK TO WAIT AFTER INTERRUPT
                                        ;E.G., LINE CLOCK

;DISPLAY FILE
117124  A:      POINT+LPOFF+INT4+BLKOFF+LINE0 ;POINT MODE, LIGHT PEN DISABLE
                                        ;INTENSITY 4; NO BLINK, SOLID LINES
000500  PX:     500                     ;MOVE BEAM TO X = 500(8) = 320.
000500  PY:     500                     ;AND           Y = 500(8) = 320.
110000          LONGV                   ;LONG VECTOR MODE
040200          200+INTX                ;L1:DELTA X = 200, INTENSIFY
000000          0                       ;   DELTA Y = 0
040000          0+INTX                  ;L2:DELTA X = 0   , INTENSIFY
000200          200                     ;   DELTA Y = 200
060200          200+INTX+MINUSX         ;L3:DELTA X =-200, INTENSIFY
000000          0                       ;   DELTA Y = 0
040000          0+INTX                  ;L4:DELTA X = 0   , INTENSIFY
020200          200+MINUSY              ;   DELTA Y =-200
160000          DJMP
??????  TARGET: A                       ;ABSOLUTE ADDRESS OF "A"
                .END START
```

Еще раз отметим, что у всех инструкций MSB равен 1, а у данных — 0.

9.1.3. Динамическая манипуляция дисплейным файлом

Если необходимо заставить квадрат медленно двигаться по экрану, то для этого применяются два способа. Первый состоит в том, что CPU увеличивает содержимое ячеек PX и PY через соответствующие интервалы времени, тогда как DPU, непрерывно сканируя дисплейный файл, снова и снова исполняет его. Второй способ заключается в создании второго дисплейного файла квадрата с медленно перемещающимися сторонами. И если в ячейку памяти с меткой «TARGET» (смотри программу выше) помещается адрес начала нового дисплейного файла, то при очередном выполнении инструкции DJMP в старом управление будет передано на начало нового файла. Старый файл может теперь модифицироваться. После этого на него опять передается управ-

ление, т. е. процесс повторяется. Последний способ, обычно называемый «*буферированием двух файлов отображений,* предпочтительнее, особенно если файл должен изменять структуру рисунка между кадрами (например, когда мы пытаемся «оживить» изображение). Первый способ используется в параграфе 9.1.5, в программе, иллюстрирующей работу системы светового пера.

9.1.4. Графические подпрограммы

Комбинируя линейные сегменты, пользователь может нарисовать сложные объекты. И наоборот, произвольные картины могут быть представлены элементами, более сложными, чем линии. Часто большинство из этих элементов оказывается идентичным. Так, например, шахматная доска состоит из квадратов, а фасад дворца — из повторяющихся идентичных окон и колонн. Поэтому будет экономным, если мы однажды определим такую деталь (картины), а затем, при движении луча по экрану к требуемой точке, будем вставлять эту деталь в нужные места. Так изображаются символы, определения которых размещены в ROM. Оборудование сложных графических дисплеев (например, VS-60) обладает свойствами, аналогичными CPU, когда он выполняет свои подпрограммы; это оборудование имеет регистры и стек. Однако для VT-11 при использовании подпрограмм требуется кооперирование с CPU.

По инструкции DSTOP графический процессор моментально останавливается. Подпрограмма обработки прерывания по DSTOP загружает DPC адресом дисплейного файла, обеспечивающего «прорисовку» деталей (адрес файла находится в памяти, сразу за DSTOP), и снова инициирует работу DPU. Выход из подпрограмм дисплея обеспечивается командой DSTOP, за которой следует нуль. Лучше всего это видно на примере. Квадраты изображаются на экране, начиная с левого нижнего угла, координаты которого (100, 100), (300, 300) и (750, 750).

```
;GRAPHICS SUBROUTINES VIA DSTOP
START:  MOV     #DSUB,@#320     ;ADDRESS OF DSTOP ISR
        MOV     #200,@#322      ;PRIORITY OF DSTOP ISR
        MOV     #STACK,R5       ;SET UP R5 STACK FOR ISR
;
        MOV     #A,@#DPC        ;START DISPLAY PROCESSOR
WAIT:   WAIT                    ;CPU ONLY REQUIRED DURING ISR
        BR      WAIT            ;AFTER DSTOP INTERRUPT, BACK TO WAIT
;
        .BLKW   10.             ;SPACE FOR R5 STACK
STACK:  .WORD   0
                                ;DISPLAY FILE
A:      POINT+LPOFF+INT4+LINE0,100.,100. ;SAME AS ORIGINAL
        DSTOP
        SQUARE                  ;ADDRESS OF SQUARE DISPLAY FILE SUBR
        POINT,300.,300.         ;MOVE BEAM TO (300,300)
        DSTOP
        SQUARE                  ;THE SQUARE AGAIN
        POINT,750.,750.         ;MOVE BEAM TO (750,750)
        DSTOP
        SQUARE                  ;AND AGAIN
        DJMP.
        A                       ;END OF MAIN DISPLAY FILE
```

```
;
SQUARE: LONGV                          ;LONG VECTOR MODE
        .                              ;INSTRUCTIONS TO DRAW SQUARE
        .                              ;AS PREVIOUSLY
        DSTOP                          ;EFFECTS DISPLAY RTS
        0
;
DSUB:   TST     @DPC                   ;IF DPC POINTS TO "0", DISPLAY RTS
        BEQ     DRTS                   ;YES!
;
;NO, IMPLEMENT DISPLAY SUBROUTINE
;
        MOV     @#DPC,-(R5)            ;SAVE DPC CONTENTS OF HALTED DPU
        ADD     #2,(R5)                ;RETURN ADDR FROM SUBROUTINE = @#DPC+2
        MOV     @DPC,@#DPC             ;MOVING SUBROUTINE STARTING ADDRESS INTO
                                       ;       DPC RESTARTS DPU AT THAT ADDRESS
        RTI                            ;DPU DOING SUBROUTINE, RELEASE CPU
;
;YES, IMPLEMENT DISPLAY RETURN FROM SUBROUTINE
DRTS:   MOV     (R5)+,@#DPC            ;MOVING RETURN ADDRESS FROM SUBROUTINE
                                       ;INTO DPC RESTARTS DPU AT THAT ADDRESS
        RTI
;
        .END START
```

Картина, получающаяся в результате выполнения программы, показана на рис. 9.1. Изображение этой детали можно выполнить желаемое число раз. Увеличение памяти для файла отображения будет незначительным, причем оно *не зависит от сложности рисуемой детали.*

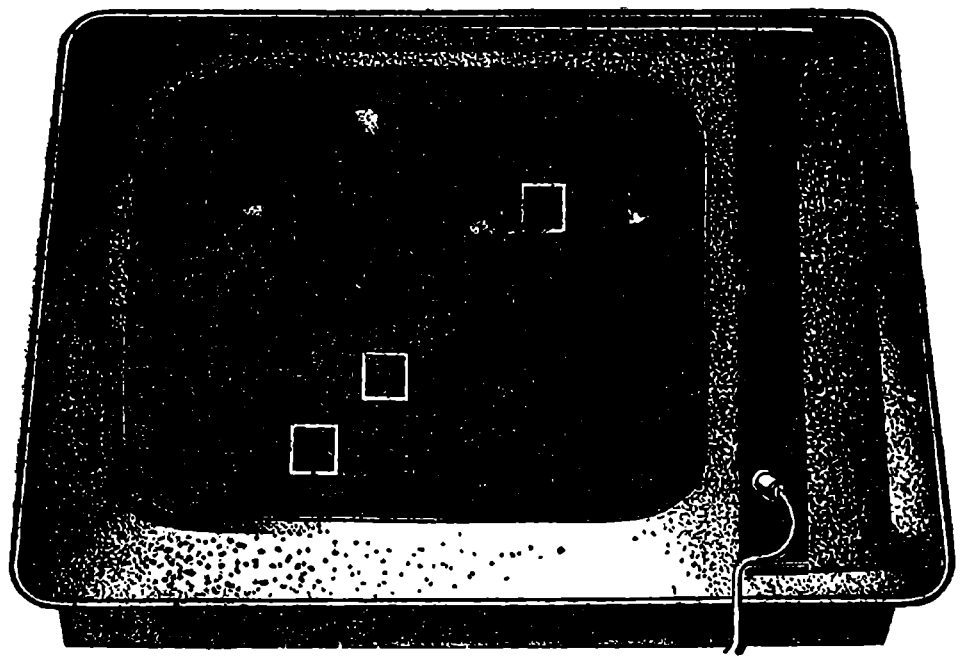


Рис. 9.1. Выход программы графического изображения

9.1.5. Работа со световым пером

Для непрофессионала работа со световым пером кажется часто чем-то магическим. Световое перо содержит фотодиод, который реагирует на свет лишь в малой спектральной области; поэтому оно будет выдавать импульс тогда и только тогда, когда позиция луча на экране совпадает с позицией, на которую указывает световое перо. Существуют два множества информации, которые могут воздействовать на программу обработки прерываний (ISR) светового пера при ее взаимодействии с картиной. Если импульс останавливает DPU, то, используя PC, можно определить адрес инструкции, ответственной за часть картины, навлек-

шей на себя импульс светового пера. Это один путь воздействия на программу обработки прерываний. Во-вторых, DPU имеет два регистра (их адреса 172004 и 172006), в которых хранится «позиция» остановленного луча. Выполнив соответствующие действия, ISR снова передает управление DPU, возобновляя его работу. Мы опять проиллюстрируем эту концепцию на коротком примере. Здесь небольшой квадрат будет следовать за световым пером по экрану.

```
;LIGHT PEN PROGRAM
START:  MOV     #LPSERV,@#324   ;LIGHT PEN ISR ADDR
        MOV     #340,@#326      ;PRIORITY OF LIGHT PEN ISR
        MOV     #A,@#DPC        ;START DISPLAY PROCESSOR
WAIT:   WAIT                    ;CPU NOW IDLE
        BR      WAIT
;
A:      POINT+LPON+INT4+BLKOFF+LINE0    ;AS BEFORE, BUT LIGHT PEN ENABLED
PX:     500.
PY:     380.
        LONGV                   ;DRAW A SQUARE, AS ABOVE
        200+INTX,0,0+INTX,200,200+INTX+MINUSX,0,0+INTX,200+MINUSY
        DJMP
        A
;DPU IS STOPPED WHEN LIGHT PEN HIT OCCURS
LPSERV: MOV     @#172004,R1     ;GET "X" DPU REGISTER CONTENTS
        BIC     #176000,R1      ;UPPER 6 BITS ARE UNDEFINED
        MOV     @#172006,R2     ;GET "Y" DPU REGISTER CONTENTS
        BIC     #176000,R2      ;UPPER 6 BITS ARE UNDEFINED
;
        MOV     PX,R3           ;GET X POS OF LOWER LEFT CORNER OF SQUARE
        MOV     PY,R4           ;GET Y POS OF LOWER LEFT CORNER OF SQUARE
;FIND OUT WHICH SIDE OF SQUARE HAD "HIT" BY SERIES OF COMPARISONS
        CMP     R1,R3
        BNE     N1
LEFT:   SUB     #2,PX           ;LEFT SIDE HIT, MOVE SQUARE LEFT
        BR      OUT
N1:     CMP     R2,R4
        BNE     N2
BOTTOM: SUB     #2,PY           ;BOTTOM HIT, MOVE SQUARE DOWN
        BR      OUT
N2:     ADD     #200,R4
        CMP     R2,R4
        BNE     RIGHT
TOP:    ADD     #2,PY           ;TOP HIT (BY DEFAULT), MOVE SQUARE UP
        BR       OUT
RIGHT:  ADD     #2,PX           ;RIGHT SIDE HIT, MOVE SQUARE RIGHT
OUT:    MOV     #1,@#DPC        ;LOADING DPC WITH "1" RESTARTS DPU
                                ;                   WHERE IT STOPPED
        RTI
        .END START
```

Читатель, проследив «логику» программы, может сам убедиться в этом. Обычно квадраты делаются очень маленькими, а световые перья имеют форму восьмиугольников (почему?). Тем не менее такие простые программы, как эта, являются ядром таких способов графического изображения, как «гибкая привязка» (за световым пером движется один конец прямой линии) и «рисование» (световое перо позади себя оставляет след).

9.2. ВЗАИМОДЕЙСТВИЕ CPU, DPU И A/D ПРИ ОБРАБОТКЕ ЦИФРОВОГО СИГНАЛА

Графический вывод является неоценимым средством визуального подтверждения работы различных алгоритмов. В следующем примере несколько модулей обработки цифрового сигнала (A/D)

комбинируются с одной из программ дисплея для того, чтобы создать «цифровой осциллоскоп». На экране дисплея *постоянно отображаются изменяющийся сигнал и его амплитудный спектр.*

9.2.1. Спектральный анализ посредством быстрого преобразования Фурье

Известно, что сигнал, имеющий частоту W Hz, для полного сохранения должен «отбираться» с частотой $2W$ в секунду. Если скорость выборки определена, то сигнал, поступающий с частотой W Hz, должен быть сильно «разбавлен» частотой, большей, чем W. Это делается для того, чтобы избежать искажений. Каждый мог наблюдать такие искажения в ковбойских фильмах; как только дилижанс начинает ускорять движение, колеса (их движение отбирается со скоростью 24 кадра в секунду) сначала вращаются быстрее, затем останавливаются, и, наконец, вращаются в обратном направлении.

Если $x(n)$ есть выборки реального сигнала (интервал выборки $=$ T с), поступающего на вход дискретного преобразования Фурье (DFT), то $X(k)$, являющаяся выходом DFT, интерпретируется следующим образом.

$X(k)$ есть выход цифрового фильтра в момент времени $(N-1)T$ с частотой ответа

$$|H_k(f)| = |\sin(\pi fTN)/\sin[\pi(fT - k/N)]|,$$

если на вход цифрового фильтра поступает сигнал выборки $x(nT)$, начиная с момента времени $t = 0$, т. е. $n = 0$. Тогда DFT можно представить как некоторое количество связанных между собой фильтров с основными частотами k/NT и величиной ответа $|\sin x/x|$. *Так, для реального сигнала*

$$x_p(n) = \cos(2\pi pn/N),\ n = 0, \ldots, N-1 =$$
$$= 0{,}5^*[\mathrm{EXP}(j\,2\pi pn/N) + \mathrm{EXP}(-j\,2\pi pn/N)],\ p < N/2,$$
$$X(k) = 0{,}5^*\left[\sum_{n=0}^{N-1}\mathrm{EXP}(-j\,2\pi n[k-p]/N) + \sum_{n=0}^{N-1}\mathrm{EXP}(-j\,2\pi n[k+p]/N\right].$$

Можно показать, что первое суммирование даст N для $k = p$ и нуль в других случаях; второе суммирование дает N при $k = N - p$ и нуль в других случаях. Это значит, что $X(k) = N/2$, если $k = p$ или $k = N - p$. Подобно этому, если

$$x_p(n) = \sin(2\pi pn/N),$$

то $X(k) = -jN/2(k = p)$ или $jN/2(k = N - p)$.
И окончательно, если

$$x_p(n) = a_p\cos(2\pi pn/N) + b_p\sin(2\pi pn/N) = \left(a_p^2 + b_p^2\right)^{0,5}\cos(2\pi pn/N - \theta),$$
$$\theta_p = \mathrm{tg}^{-1}(b_p/a_p),$$

тогда

$$|X(k)| = N/2\left(a_p^2 + b_p^2\right)^{0,5},\ k = p \quad \text{или} \quad k = N - p$$

и

$$\underline{/X(k)} = \mathrm{tg}^{-1}\left(\mathrm{Im}\,[X(k)]/\mathrm{Re}\,[X(k)]\right) =$$
$$= \theta_p,\ k = p,$$
$$= -\theta_p,\ k = N - p.$$

Отсюда следует, что периодические сигналы, с периодом выборок N, разлагаются в ряды Фурье: $X(k)$ есть коэффициенты дискретного комплексного ряда Фурье.

Однако так как $\exp(+j2\pi p) = \cos(2\pi p) + j\sin(2\pi p) = 1$, а p есть целое, легко показать, что оба уравнения, определяющие DFT, есть периодические функции от N. Это значит, что $x(n + N) = x(n)$ и $X(k + N) = X(k)$. Тогда для *произвольной* последовательности $x(n)$ выход DFT есть преобразование периодической последовательности, один из периодов которого идентичен $x(n)$. Другими словами, DFT любую последовательность воспринимает так, как если бы она была одним периодом этой периодической последовательности.

С точки зрения время/память, когда мы имеем дело со спектральным анализом реальных сигналов, выгодно применять два специальных алгоритма:

а. **Преобразование одного N-точечного комплексного преобразования в два N-точечных действительных преобразования.** Помещаем N выборок реальной последовательности $x(n)$ в действительную часть и N выборок другой реальной последовательности $y(n)$ в мнимую часть N-точечного комплексного массива $z(n)$. Чтобы получить $Z(k)$, $k = 0, \ldots, N - 1$, выполняется N-точечное преобразование FFT. После этого преобразования $x(n)$ и $y(n) - X(k)$ и $Y(k)$ могут быть получены следующим образом:

$$X(k) = [Z^*(N - k) + Z(k)]/2$$

и

$$Y(k) = j\,[Z^*(N - k) - Z(k)]/2,\quad k = 0,\ 1,\ \ldots,\ N/2 - 1.$$

«*» обозначает комплексно-сопряженные числа и $Z(N) = Z(0)$.

б. **Преобразование одного N-точечного комплексного преобразования в одно 2N-точечное действительное преобразование.** Если выборка $x(n)$ получается из четных выборок $c(n)$, $n = 0, \ldots, 2N - 1$, а $y(n)$ по-прежнему нечетная (как и выше), то первое вычисление $X(k)$ и $Y(k)$ производится по первому алгоритму, а затем

$$C(k) = [X(k) + Y(k)^* W^{-k}]/2,\quad k = 0, \ldots, N - 1,$$

где $W = \exp(j2\pi/2N)$.

9.2.2. Цифровой осциллоскоп PDP-11/VT-11 и модульный подход к конструированию программного обеспечения

В цифровом осциллоскопе используются такие понятия, как *двойная буферизация, управление на высоком уровне* и *модульное построение программного обеспечения.* Это можно увидеть при анализе управляющей программы:

```
C       LINKING INSTRUCTIONS FOR MODULES:
C       SCOPET=SCOPET/F,TRIGGR,WINDOW,FTMUF7/C
C       BUTMUF,CTOR7S,DOUBLE,COMPU3,SCOPED
C
        INTEGER A(256),B(256),NTAB(256),N1(300),N2(300)
        COMMON /XX/A,B,NTAB
        PI=3.1415926
C       CREATE WINDOW TABLE
        DO 1 I=1,256
        XI=I-1
1       NTAB(I)=(2.**15-1.)*(0.54-0.46*COS(2.*PI*XI/256.))
C
        NFLAG=0
        NSMPL=260
        NTICK=125
C       START TO SAMPLE INTO BUFFER 1
        CALL TRIGGR(N1(1),NSMPL,NTICK)
C       WAIT, USER GIVES <CR>
        PAUSE
C
C       START TO SAMPLE INTO BUFFER 2
11      CALL TRIGGR(N2(1),NSMPL,NTICK)
C
C       PROCESS BUFFER 1
        CALL WINDOW(N1(1))
        CALL FTMUF7
        CALL CTOR7
        CALL DOUBLE
C
C       SWITCH TO DISPLAY FILE 1
        CALL SCOPE1(N1(1),NFLAG)
        NFLAG=1
C
C       START TO SAMPLE INTO BUFFER 1
        CALL TRIGGR(N1(1),NSMPL,NTICK)
C
C       PROCESS BUFFER 2
        CALL WINDOW(N2(1))
        CALL FTMUF7
        CALL CTOR7
        CALL DOUBLE
C
C       SWITCH TO DISPLAY FILE 2
        CALL SCOPE1(N2(1),NFLAG)
        GO TO 1
        END
```

В буфер 1 по прерываниям поступают выборки волновых колебаний, в то время как CPU обрабатывают буфер 2. Одновременно результаты обработки предыдущей фазы отображаются графическим дисплеем. Буферы затем переключаются, и последовательность операций повторяется.

Мы будем анализировать модули приведенной программы последовательно.

TRIGGER. Желательно иметь устойчивый образ отобранных волновых колебаний. Этого можно добиться, если выборку начинать

сразу после перехода луча через нуль в положительную область. Это значит, что предыдущая выборка была отрицательной, а данная — положительной. Как только эта ситуация обнаружена с помощью подпрограммы обработки прерываний «SERV1», сразу производится замена адреса подпрограммы обработки прерываний на «SERV2». Процессор продолжает обработку предыдущего буфера, в то время как наполняется текущий.

```
        .TITLE  TRIGGR
        .GLOBL  TRIGGR TYPE
FLG:    0
TYPE:   1
        ADVEC1=340                      ;A/D ISR VECTOR
        ADVEC2=342                      ;A/D ISR VECTOR
        ADSR=170400                     ;A/D STATUS REGISTER
        ADBR=170402                     ;A/D BUFFER
        CLKSR=170404                    ;CLOCK STATUS REGISTER
        CLKBR=170406                    ;CLOCK BUFFER
TRIGGR:
        MOV     2(R5),ADDR              ;ADDRESS OF OUTPUT SAMPLE BUFFER
        MOV     @4(R5),R0               ;NUMBER OF SAMPLES TO BE GATHERED
        MOV     @6(R5),R1               ;T = # OF USEC BETWEEN SAMPLES
        NEG     R1
        MOV     R1,@#CLKBR              ;-T INTO CLOCK BUFFER
        MOV     #SERV1,@#ADVEC1         ;SET UP A/D ISR VECTOR
        MOV     #340,@#ADVEC2           ;PRIORITY 7
        MOV     #140,@#ADSR             ;A/D STATUS: START ON CLOCK
                                        ;OVERFLOW, INTERRUPT WHEN DONE
        CLR     R1
        CLR     R2
        CLR     FLG                     ;BUFFER SWITCHING FLAG
        MOV     #423,@#CLKSR            ;CLOCK STATUS: REPEATED INTERVAL
                                        ;              1 USEC TICKS, GO!
        MOV     #256.,COUNT             ;256 SAMPLES GATHERED
WAITL:
        WAIT                            ;WAIT FOR A/D DONE INTERRUPT
        TST     FLG     ;STILL WAITING FOR POSITIVE GOING ZERO CROSSING?
        BEQ     WAITL                   ;YES!
        RTS     PC                      ;NO, RETURN TO MAIN
;ISR FOR POSITIVE GOING ZERO CROSSING DETECTION
SERV1:
        MOV     @#ADBR,R2               ;PICK UP A/D SAMPLE
        SUB     #1000,R2                ;CONVERT TO 2'S COMPLEMENT
        BLT     OUT                     ;IF SAMPLE < 0, RETURN TO WAIT
        TST     R1                      ;IF > 0, TEST PREVIOUS SAMPLE
        BGE     OUT                     ;IF IT WAS > 0, RETURN TO WAIT
        MOV     #1,FLG                  ;PRESENT > 0, PREVIOUS < 0! GO!
        MOV     #SERV2,@#ADVEC1         ;NEW ISR TO GATHER REST OF SAMPLES
        MOV     @#ADBR,@ADDR            ;SAMPLE INTO BUFFER
        ADD     #2,ADDR                 ;BUMP BUFFER POINTER
        DEC     COUNT                   ;DECREMENT SAMPLE COUNT
        RTI                             ;RTI
OUT:
        MOV     R2,R1                   ;SAVE THIS SAMPLE FOR NEXT TIME
        RTI
;ISR FOR BACKGROUND SAMPLE GATHERING.  NO REGISTERS USED.
SERV2:
        MOV     @#ADBR,@ADDR            ;SAMPLE INTO BUFFER
        ADD     #2,ADDR                 ;BUMP BUFFER POINTER
        DEC     COUNT                   ;DECREMENT SAMPLE COUNT
        BEQ     STOP                    ;ENOUGH SAMPLES?
        RTI                             ;NO.
STOP:   CLR     @#CLKSR                 ;YES. TURN OFF CLOCK, A/D
        RTI
COUNT:  0
ADDR:   0
        .END
```

WINDOW. Входная выборка сначала должна быть преобразована в форму с дополнением до двух (SUB #1000, R4), а затем

умножена на положительную «функцию окна» («window function»),

$$w(i) = (2{**}15 - 1){*}[0{,}54 - 0{,}46{*}\cos(2\pi[i - 1]/N)], \quad i = 1, 2, \ldots, N.$$

где $N = 256$ (в этом примере). Применение WINDOW обеспечивает улучшенные спектральные оценки. Заметим, что сохраняются две версии сигнала, обработанного «функцией окна»; немасштабируемая версия для дисплея (MOV R4, (R0)+) и масштабируемая версия для ввода в алгоритмы быстрого преобразования Фурье (FFT) (ASHC #3, R4 и MOV R4, (R3)+).

```
        .TITLE  WINDOW
        .GLOBL  WINDOW,TYPE
        .CSECT  XX                  ;LABELLED COMMON
        A=.
        B=.+512.
        TABLEN=.+512.
        .CSECT
WINDOW:
        MOV     2(R5),R0            ;GET THE ADDRESS OF ARGUMENT #1
        MOV     #A,R3               ;GET THE ADDRESS OF THE A ARRAY


;       NARROW BAND
NARROW:
        MOV     #TABLEN,R2          ;GET THE ADDRESS OF THE WINDOW TABLE
        MOV     #256.,R1            ;INITIALIZE THE LOOP COUNTER
;       DO THE WINDOWING USING A 256 POINT WINDOW (I.E., N = 256)
LOOPN:
        MOV     (R0),R4             ;GET SAMPLE
        SUB     #1000,R4            ;ELIMINATE BIAS OF AR-11 A/D
        MUL     (R2)+,R4            ;MULTIPLY BY WINDOW FUNCTION
        MOV     R4,(R0)+            ;BACK TO BUFFER FOR DISPLAY USE
        ASHC    #3,R4               ;SHIFT LEFT 3 PLACES
        MOV     R4,(R3)+            ;INTO ANOTHER BUFFER FOR FFT USE
        SOB     R1,LOOPN
;
        RTS     PC                  ;RETURN
        .END
```

FTMUF7. FTMUF7 — это 128-точечное FFT.

CTOR7/DOUBLE/COMPU3. CTOR7 вычисляет $2 * X(k)$, $2 * Y(k)$, как в 9.2.1а, а DOUBLE использует COMPU3 для вычисления $2 * C(k)$, как в параграфе 9.2.1б. Коэффициент 2 применяется для увеличения скорости и удобства. Читатель может заметить также, что в CTOR7 применяется автоинкрементный/автодекрементный режим адресации для получения оптимального с точки зрения времени и памяти кода. Алгоритм, адресные режимы и структуры данных подобраны с учетом особенностей работы каждого из этих компонентов.

```
        .TITLE CTOR7
        .GLOBL CTOR7
        .CSECT XX                   ;LABELLED COMMON
        A=.
        B=.+512.
        .CSECT
CTOR7:
        MOV #B,R0                   ;START OF B BUFFER (INPUT)
        MOV #B+ 508.,R1             ;"END" OF B BUFFER (INPUT)
        MOV #A,R2                   ;START OF A BUFFER (INPUT/OUTPUT)
```

```
;SPECIAL CODE FOR POINTS 0,1
        ASL (R0)+               ; Re[X(0)] ← 2*Re[Z(0)]
        MOV (R0),(R2)
        ASL (R2)+               ; Re[Y(0)] ← 2*Im[Z(0)]
        CLR (R0)+               ; Im[X(0)] ← 0
        CLR (R2)+               ; Im[Y(0)] ← 0
        TST (R2)+               ; POINT TO Im[Y(1)]
        MOV (R1),(R2)
        SUB (R0),(R2)           ; Im[Y(1)] ← Re[Z(N-1)]-Re[Z(1)]
        ADD (R1)+,(R0)+         ; Re[X(1)] ← Re[Z(N-1)]+Re[Z(1)]
        MOV (R1),-(R2)
        ADD (R0),(R2)           ; Re[Y(1)] ← Im[Z(1)]+Im[Z(N-1)]
        SUB (R1),(R0)+          ; Im[X(1)] ← Im[Z(1)]-Im[Z(N-1)]
        MOV #B+510.,R1
        MOV #A+  4,R2
        MOV #   62.,R3
        MOV #    6,R4

;
LOOP:   SUB R4,R1               ;BUMP POINTER
        ADD R4,R2               ;BUMP POINTER
        MOV (R1),(R2)
        SUB (R0),(R2)           ; Im[Y(k)] ← Re[Z(N-k)]-Re[Z(k)]
        ADD (R1)+,(R0)+         ; Re[X(k)] ← Re[Z(N-k)]+Re[Z(k)]
        MOV (R1),-(R2)
        ADD (R0),(R2)           ; Re[Y(k)] ← Im[Z(k)]+Im[Z(N-k)]
        SUB (R1),(R0)+          ; Im[X(k)] ← Im[Z(k)]-Im[Z(N-k)]
        SOB R3,LOOP

        RTS PC
        .END
;
        .TITLE DOUBLE
        .GLOBL DOUBLE,COMPUT,TYPE,TABLE
        .CSECT XX               ;LABELLED COMMON
        A=.
        B=.+  1000
        .CSECT
DOUBLE:
        TST TYPE
        BNE 1$
        RTS PC
1$:
        ADD A,B
        ASR B
        ADD A+2,B+2
        ASR B+2
        MOV #    63.,R4
        MOV #A,R2
        MOV #TABLE+2,R3
        MOV #B,R5
        JSR PC,COMPUT
        NEG B+   402
;
        RTS PC
TABLE:
    .WORD      0, 77777,  1444, 77765,  3107, 77727,  4552, 77646
    .WORD   6213, 77541,  7653, 77410, 11307, 77234, 12741, 77034
    .WORD  14370, 76611, 16013, 76342, 17431, 76051, 21043, 75534
    .WORD  22447, 75174, 24046, 74611, 25436, 74203, 27020, 73553
    .WORD  30373, 73100, 31736, 72403, 33271, 71665, 34614, 71124
    .WORD  36126, 70341, 37426, 67536, 40715, 66711, 42172, 66043
    .WORD  43434, 65154, 44663, 64245, 46077, 63316, 47277, 62347
    .WORD  50463, 61361, 51632, 60353, 52764, 57326, 54102, 56263
    .WORD  55201, 55201, 56263, 54102, 57326, 52764, 60353, 51632
    .WORD  61361, 50463, 62347, 47277, 63316, 46077, 64245, 44663
    .WORD  65154, 43434, 66043, 42172, 66711, 40715, 67536, 37426
    .WORD  70341, 36126, 71124, 34614, 71665, 33271, 72403, 31736
    .WORD  73100, 30373, 73553, 27020, 74203, 25436, 74611, 24046
    .WORD  75174, 22447, 75534, 21043, 76051, 17431, 76342, 16013
    .WORD  76611, 14370, 77034, 12741, 77234, 11307, 77410,  7653
    .WORD  77541,  6213, 77646,  4552, 77727,  3107, 77765,  1444
        .END
;
```

```
        .TITLE  COMPU3
        .GLOBL  COMPUT
        .CSECT XX                       ;LABELLED COMMON
        A=.
        B=.+1000
        .CSECT
COMPUT:
        MOV     R4,-(SP)                ;PUT COUNT ON STACK
LOOP:   ADD     #6,R2                   ;R2 GETS R2-2 EACH PASS
        ADD     #4,R3                   ;R3 OK
        ADD     #6,R5                   ;R5 GETS R5-2 EACH PASS
        MOV     R5,-(SP)                ;SAVE R5 FOR LATER
        MOV     (R2),R0
        MUL     (R3),R0                 ;R0 ← IM(A2)*COS/2
        MOV     -(R2),R4
        MUL     (R3),R4
        MOV     R4,R1                   ;R1 ← RE(A2)*COS/2
        MOV     -(R3),R4
        MUL     (R2)+,R4                ;R4 ← RE(A2)*SIN/2
        SUB     R4,R0                   ;R0 ← IM(A2)*COS/2-RE(A2)*SIN/2
        MOV     (R3)+,R4
        MUL     (R2),R4                 ;R4 ← IM(A2)*SIN/2
        ADD     R1,R4                   ;R4 ← RE(A2)*COS/2+IM(A2)*SIN/2
        MOV     (SP)+,R5                ;RESTORE R5
        ASR     (R5)                    ;SCALE IM(A1)
        MOV     (R5),(R2)
        ADD     R0,(R5)                 ;IM(A1) ← IM(A1)+R0
        SUB     R0,(R2)                 ;IM(A2) ← IM(A1)-R0
        NEG     (R2)
        ASR     -(R5)                   ;SCALE RE(A1)
        MOV     (R5),-(R2)
        ADD     R4,(R5)                 ;RE(A1) ← RE(A1)+R4
        SUB     R4,(R2)                 ;RE(A2) ← RE(A1)-R4
        DEC     (SP)
        BNE     LOOP
        TST     (SP)+                   ;POP COUNTER OFF STACK
        RTS     PC
        .END
```

SCOPE1. SCOPE1 сначала вычисляет $|C(k)|$, модуль $C(k)$. Это выполняется суммированием квадратов действительной и мнимой частей $C(k)$, а затем используется алгоритм Ньютона — Рафсона для извлечения квадратного корня. Волновые колебания, обработанные «функцией окна», отображаются с помощью линейных сегментов для соединения точек выборки, в то время как вывод спектра осуществляется вертикальными линиями. Здесь также применяется двойная буферизация. Пока отображается одна волна и спектр, для другой строится дисплейный файл. Первоначально он имеет следующую структуру:

```
DISPST:         .       ;DISPLAY FILE FOR AXES, LABELLING
                .
                .
DIS1:           DJMP
D1:             END1    ;WAVEFORM/SPECTRUM DISPLAY FILE #1
                .
                .
                .
END1:           DJMP
                DIS2
;
DIS2:           DJMP
D2:             END2
                .       ;WAVEFORM/SPECTRUM DISPLAY FILE #2
                .
                .
END2:           DJMP
                DISPST
```

Читатель может получить некоторое удовлетворение от знания того, что последовательность команд

```
MOV #END2,D2
MOV #D1+2,D1
```

остановит отображение буфера 2 и *начнет* отображение буфера 1, в то время как

```
MOV #END1,D1
MOV #D2+2,D2
```

остановит отображение буфера 1 и *начнет* отображение буфера 2.

Следующая ниже программа получает комплексные выборки из дискретного преобразования Фурье (DFT), вычисляет модуль каждой из них (т. е. корень квадратный из суммы квадратов действительной и мнимой частей), а затем создает дисплейный файл и амплитудный спектр для каждого вводимого сигнала. Сигнал, обработанный функцией окна, отображается на дисплее. Для интерполяции используются прямые линии («соединить точки»), а спектр представлен прямыми линиями соответствующей длины.

```
.GLOBL SCOPE1 DIS1 D1 END1 DIS2 D2 END2
.CSECT XX               ; LABELLED COMMON
A=.
B=.+1000
.CSECT
BLKOFF=20
POINT=   114000
LONGV=   110000
SHORTV=  104000
INT7=    3600
CHAR=    100000
LPON=    140
LPOFF=   100
LINE0=   4
DJMP=    160000
INTX=    40000
DNOP=    164000
MINUSX=  20000
DPC=     172000


  SCOPE1:
          MOV     2(R5),ARG         ;ADDRESS OF TIME WAVEFORM BUFFER
          TST     @4(R5)            ;IS IT FIRST CALL?
          BNE     GO
  ;START DISPLAY AND INITIALIZE FLAG FIRST TIME AROUND
          MOV     #DISPST,@#DPC     ;YES,START DISPLAY
          MOV     #-1,FLAG          ;BUFFER FLAG
  ;
  GO:     NEG     FLAG              ;FLAG WILL BE +1 ON FIRST CALL
          CLR     BUF1              ;D.C. OUTPUT = 0
          MOV     #B+4,R5           ;INPUT 1'ST HARMONIC ADDRESS
          MOV     #BUF1+2,OUTADD    ;OUTPUT 1'ST HARMONIC ADDRESS
          MOV     #64.,-(SP)        ;STORE COUNT ON STACK FOR EFFICIENCY
```

```
;
;CALCULATE POWER SPECTRUM/AMPLITUDE SPECTRUM
;A) FIRST HALF OF ARRAY
LOOPA:
        MOV     (R5)+,R0        ;REAL
        MUL     R0,R0           ;REAL*REAL
        MOV     (R5)+,R2        ;IMAG
        MUL     R2,R2           ;IMAG*IMAG
        ADD     R1,R3
        ADC     R2              ;SUM SQUARES "K" IN R2,R3
        ADD     R0,R2
        JSR     PC,NEWTON       ;GET SQUARE ROOT OF R2,R3 IN R4
        MOV     R4,@OUTADD      ;OUTPUT SQUARE ROOT
        ADD     #2,OUTADD       ;BUMP ARRAY ADDRESS
        DEC     (SP)            ;DECREMENT COUNTER
        BGT     LOOPA           ;BRANCH IF NON ZERO
;
        MOV     #63.,(SP)       ;RELOAD COUNTER
        MOV     #A+252.,R5
;B) SECOND HALF OF ARRAY, SAME STEPS AS FIRST
LOOPB:
        MOV     -(R5),R0        ;IMAG
        MUL     R0,R0           ;IMAG*IMAG
        MOV     -(R5),R2        ;REAL
        MUL     R2,R2           ;REAL*REAL
        ADD     R1,R3
        ADC     R2              ;SUM SQUARES "K" IN R2,R3
        ADD     R0,R2
        JSR     PC,NEWTON       ;GET SQUARE ROOT OF R2,R3 IN R4
        MOV     R4,@OUTADD      ;OUTPUT SQUARE ROOT
        ADD     #2,OUTADD       ;BUMP ARRAY ADDRESS
        DEC     (SP)            ;DECREMENT COUNTER
        BGT     LOOPB           ;BRANCH IF NON-ZERO
        TST     (SP)+           ;POP COUNT
;
        BR      OUTPUT
;
;USE NEWTON-RAPHSON ALGORITHM TO CALCULATE SQUARE ROOT
NEWTON:
        .....program of Section 9.2 :....
;
;START OBTAINING VALUES FOR SPECTRUM DISPLAY
OUTPUT:
        MOV     ARG,R2          ;INPUT BUFFER ADDRESS FOR TIME WAVEFORM
        MOV     #BUF1,R4        ;ADDRESS OF AMPLITUDE SPECTRUM ARRAY
;
        TST     FLAG            ;ALTERNATE BUFFER FLAG, CHANGED EACH PASS
        BMI     NEG

POS:
        CLR     TIMFL1-2        ;TIME WAVE =0 AT T=-1 FOR DELTA TO T=0
        MOV     #TIMFL1,R5      ;OUTPUT BUFFER FOR TIME WAVEFORM
                                ;"CONNECT THE DOTS"
        MOV     #FFL1,R3        ;OUTPUT BUFFER FOR AMPLITUDE SPECTRUM
                                ;"VERTICAL LINES"
        BR      LOOP3
;
NEG:
        CLR     TIMFL2-2        ;SAME AS ABOVE FOR ALTERNATE PASSES
        MOV     #TIMFL2,R5
        MOV     #FFL2,R3
LOOP3:
;
;SPECTRAL SAMPLE--VISIBLE VERTICAL LINE
        MOV     #INTX,(R3)+     ;INTENSIFY, DEL X = 0 FOR SPECTRAL SAMPLE
        ASR     (R4)
        ASR     (R4)
        MOV     (R4),(R3)+      ;DEL Y = SPECTRAL VALUE/4, ALWAYS POSITIVE
;
;INVISIBLE LINE FROM TOP OF PRESENT LINE TO BOTTOM OF NEXT LINE
        MOV     #6,(R3)+        ;DEL X =6 TO BASE OF NEXT SPECTRAL LINE
        MOV     (R4)+,(R3)      ;
        BIS     #MINUSX,(R3)+   ;DEL Y = -DEL Y TO BASE OF NEXT LINE
;
;DO 2 SIGNAL SAMPLES--CONNECT THE DOTS
        MOV     #2,COUNT
```

```
LOOP4:
        MOV     #INTX+3,(R5)+   ;INTENSIFY X, DEL X = 3 BETWEEN SAMPLES
        ASR     (R2)            ;YNEW=SIGNAL/2
        MOV     (R2),-(SP)      ;YNEW STORED TEMPORARILY
        SUB     R0,(R2)         ;DEL Y=YNEW-YOLD TO "CONNECT THE DOTS"
        MOV     (SP)+,R0        ;YOLD NEXT=YNEW PRESENT
;
        TST     (R2)            ;IS DEL Y NEGATIVE?
        BGE     XONT20
        NEG     (R2)            ;NEGATE IF NEGATIVE
        BIS     #MINUSX,(R2)
XONT20: MOV     (R2)+,(R5)+     ;DEL Y = MAG(YNEW-YOLD),
                                ;    WITH CORRECT SIGN BIT
        DEC     COUNT
        BNE     LOOP4
;
        CMP     R4,#BUF1+254.   ;FINISHED?
        BLO     LOOP3           ;NO
;
        TST     FLAG
        BMI     NEG2
POS2:   MOV     #END2,D2        ;STOP DISPLAY FILE #2
        MOV     #D1+2,D1        ;START DISPLAY FILE #1
        RTS     PC
;
NEG2:   MOV     #END1,D1        ;STOP DISPLAY FILE #1
        MOV     #D2+2,D2        ;START DISPLAY FILE #2
        RTS     PC
;
```

```
COUNT:  0
OUTADD: 0
FLAG:   0
ARG:    0
TEMP:   0
N:      0
BUF1:   .BLKW   256.
;
;DISPLAY FILE WITH DOUBLE BUFFER
;
DISPST: POINT!LINE0!LPOFF!INT7!BLKOFF,100.,0 ;BEAM TO (100,0)
        LONGV,0+INTX,767.       ;VERTICAL LINE 767 UNITS
        POINT,100.,576.         ;BEAM TO (100,576)
        LONGV,768.+INTX,0       ;HORIZONTAL LINE 768 UNITS, TIME AXIS
        POINT,100.,20.          ;BEAM TO (100,20)
        LONGV,768.+INTX,0       ;HORIZONTAL LINE 768 UNITS, FREQ AXIS
;
DIS1:   DJMP                    ;DISPLAY FILE 1
D1:     END1
        POINT,100.,576.         ;BEAM TO (100,576) (TIME WAVEFORM START)
        LONGV,MINUSX+3,0
TIMFL1: .BLKW   508.            ;TIME WAVEFORM VECTORS GO HERE
        POINT,100.,20.          ;BEAM TO (100,20)  (SPECTRUM START)
        LONGV
FFL1:   .BLKW   508.            ;SPECTRUM VECTORS GO HERE
END1:   DJMP
        DIS2
;
DIS2:   DJMP                    ;DISPLAY FILE 2
D2:     END2
        POINT,100.,576.         ;BEAM TO (100,576) (TIME WAVEFORM START)
        LONGV,MINUSX+3,0
TIMFL2: .BLKW   508.            ;TIME WAVEFORM VECTORS GO HERE
        POINT,100.,20.          ;BEAM TO (100,20)  (SPECTRUM START)
        LONGV
FFL2:   .BLKW   508.            ;SPECTRUM VECTORS GO HERE
END2:   DJMP
        DISPST
;
        .END
```

БИБЛИОГРАФИЯ

Анализ аналого-цифровых преобразователей приводится в деталях в Sheingold D H. (1972), в то время как в Newman W. и Sproull R. (1973) и Giloi W. K. (1978) обсуждаются оборудование интерактивных графических процессоров, структуры отображаемых данных, алгоритмы преобразования картин и языки графических построений. Инструкция по VT-11 фирмы DEC описывает оборудование VT-11 и дальнейшие возможности по расширению аспектов программирования. В работе Flanagan J. L. (1972) изложены стандартные приемы анализа, синтеза и восприятия речи, а книга Flanagan J. L., Rabiner L. R. (1973) содержит перепечатанные классические статьи по вопросам синтеза речи. Цифровая обработка сигналов речи изложена в Rabiner L. R., Schafer R. W. (1978); анализ/синтез речи посредством линейного предсказывающего кодирования полностью объясняется в Markel J. D., Gray A. H. (1976). В статье Morris L. R. (1978) содержится полное программное обеспечение для спектропрограмм, кроме того, Morris L. R., Mudge J. C. (1978) описывают эксперименты микропрограммирования для 11/60. В работе Morris L. R., Allan D. I. (1979) описан синтезатор речи, запрограммированный для выполнения на PDP-11/60. Elovitz H. S. and all. (1976) создали способы и правила перевода с английского на VOTRAX; Broihier J. C. и Crowley M. J. воплотили этот алгоритм, написав программы на SNOBOL'е и BASIC-PLUS, что послужило основой программного обеспечения, приведенного в Morris L. R. (1978). Gagnon R. T. (1978), автор VOTRAX'а, излагает некоторые детали прибора и указывает источники, описывающие его более подробно. Wiggins R., Brantingham L. (1978) обсуждают каскадный умножитель, являющийся ядром чипов, синтезирующих речевой ввод-вывод. Moorer J. A. (1977) дает обзор аспектов создания музыкальных произведений с помощью ЭВМ; у него же имеется обширная библиография. Chamberlin H. (1977) описывает синтез музыки в ЭВМ с практической точки зрения на основе использования таблицы волн.

Глава 10

ОПЕРАЦИОННЫЕ СИСТЕМЫ

Современную мини-ЭВМ нельзя представить без операционной системы, сейчас развитыми дисковыми или ленточными операционными системами комплектуются даже самые малые микро-ЭВМ. Поэтому работа любого программиста всегда в той или иной степени связана с этой составляющей вычислительной установки.

В основном операционные системы служат для обеспечения условий взаимодействия пользователя с ЭВМ, заполняют разрыв между характеристиками технических средств и потребностями человека. Средства операционной системы поддерживают весь цикл работ по созданию программного обеспечения, а также осуществляют управление самой ЭВМ. В большинстве случаев пользователи не знакомы с принципами устройства операционных систем, их уровень знаний обычно укладывается в рамки тех обслуживающих средств, которые предоставляет система, т. е. средств подготовки текстов, трансляции, исполнения и отладки программ и файловой системы.

Для того чтобы сложилось общее представление о вычислительной системе, необходимо понять, какие компоненты ее составляют и как они организуются в единое целое. О компонентах — технических средствах и программном обеспечении — говорилось в предшествующих главах. В этой главе рассмотрим общую организующую функцию, выполняемую операционной системой. Как и прежде, нас интересуют небольшие вычислительные системы, и поэтому мы будем рассматривать соответствующие им операционные среды. Ограниченность ресурсов, которые доступны системам, работающим на мини-ЭВМ, приводит, в свою очередь, к ограниченности самих операционных систем по сравнению с системами для больших вычислительных машин. Принципы, лежащие в основе и тех и других, одни и те же, основное различие состоит в том, что в системах разного масштаба главное внимание уделяется различным системным компонентам.

10.1. ПРОСТЕЙШАЯ ВЫЧИСЛИТЕЛЬНАЯ СИСТЕМА

До последнего времени для ввода-вывода в большинстве ЭВМ применялась перфолента. Современные дисковые запоминающие устройства, например флоппи-диски, заменили перфоленту

в качестве носителя системы, хотя для некоторых целей она продолжает использоваться, в частности для диагностики. С перфолентой связана эпоха зарождения ЭВМ, программы и данные тогда вводились и выводились только при помощи перфоустройств.

В ту пору даже загрузка программ выполнялась вручную. Операционная система, хранящаяся на перфоленте, обладала возможностями, описанными ранее в гл. 8 и других главах. Она представляла собой довольно развитый пакет программного обеспечения, который давал пользователю возможность самому писать, редактировать, отлаживать, загружать и выполнять свою программу. Работа с такой системой весьма трудоемка. Имея подготовленную на перфоленте программу, пользователь должен был вручную выполнить следующие действия:

1) загрузить программу-редактор с перфоленты;
2) выполнить редактирование и получить перфоленту с текстом программы;
3) загрузить с перфоленты ассемблер;
4) выполнить трансляцию программы;
5) загрузить и выполнить полученную в результате работы ассемблера программу, хранящуюся в двоичном коде на перфоленте;
6) отладить программу, повторяя при необходимости предыдущие пять шагов решения.

Алгоритм работы с системой, описанный выше, предполагает, что в памяти ЭВМ постоянно хранятся программа первичной загрузки и абсолютный загрузчик. К сожалению, так бывает не всегда, чаще пользователю приходится самому загружать по мере надобности и эти программы.

10.2. КОМПОНЕНТЫ ДИСКОВОЙ ОПЕРАЦИОННОЙ СИСТЕМЫ

Добавление к системе *вторичного запоминающего устройства*, например диска, позволяет создавать *дисковые операционные системы* (DOS), которые представляют собой мощное средство для разработки и выполнения программ пользователей. Эти и системные программы, а также данные могут размещаться на диске или другом вторичном запоминающем устройстве и загружаться с него в память под управлением операционной системы. Вместо того, чтобы вручную загружать и перезагружать перфоленты, можно использовать DOS, возложив на нее выполнение соответствующих функций, определяемых командами. Эти команды дают не только возможность выполнять служебные операции, например загрузку, но и предоставляют средства для эффективного управления программами и системой. В табл. 10.1 приведены основные команды используемой на PDP-11 операционной системы RT-11 и описание их функций.

Очевидно, что команды представляют собой внешнее проявление операционной системы. Для полного понимания того, как они

Таблица 10.1. **Основные команды системы RT-11**

Команда	Функция
ASSIGN	Присваивает устройству имя
DATE	Вводит дату, используется для идентификации файлов
E	Анализирует содержимое указанной ячейки памяти
RUN	Загружает и выполняет указанную программу
GET	Загружает указанную программу
START	Начинает исполнение ранее загруженной программы с указанного адреса
TIME	Вводит время для системных часов
SUSPEND	Приостанавливает выполнение текущей задачи
RSUME	Возобновляет выполнение приостановленной задачи

выполняются и что реализуют, необходимо детальное рассмотрение компонент операционной системы и их организации. Поскольку одной из наиболее важных функций операционной системы является эффективное управление информационными структурами (данными и программами), необходимо ввести в рассмотрение базовую единицу хранения информации в системе. Такой единицей является *файл.*

10.2.1. Организация файлов и методы доступа

Файл — это набор *связанных записей* или элементов данных, с которым можно работать как с *одним информационным модулем.* Слово файл в самом общем смысле относится к произвольному набору информационных элементов, которые близки по назначению, форме или содержанию. Например, файлами можно назвать некоторую структуру данных, программу и даже целую систему программ, такую, как ассемблер или редактор. Для хранения файлов используются магнитные ленты или диски, поэтому устройства такого типа можно назвать *устройствами, ориентированными на хранение файлов.*

Каждое такое устройство имеет свою *файловую структуру,* определяющую методы записи, связь между элементами и каталогизацию данных; иначе говоря, она определяет организацию файлов на устройстве и методы доступа к ним. Организационное структурирование имеет важное значение в связи с тем, что файл может быть эффективно использован только в том случае, если выбранная структура соответствует специфическим потребностям пользователя. При определении структуры файла следует учитывать такие факторы, как размер, количество обращений и доступность.

Способ организации файла на носителе информации определяется тем, как пользователь предполагает его создавать и впоследствии с ним работать. Существуют три вида распределения памяти для файла: непрерывное, списковое и индексное. Пример каждого из них приведен на рис. 10.1.

В небольших вычислительных системах пользователь не имеет возможности менять что-либо в структуре организации файлов, она заранее определена для каждого типа устройств. Точно так

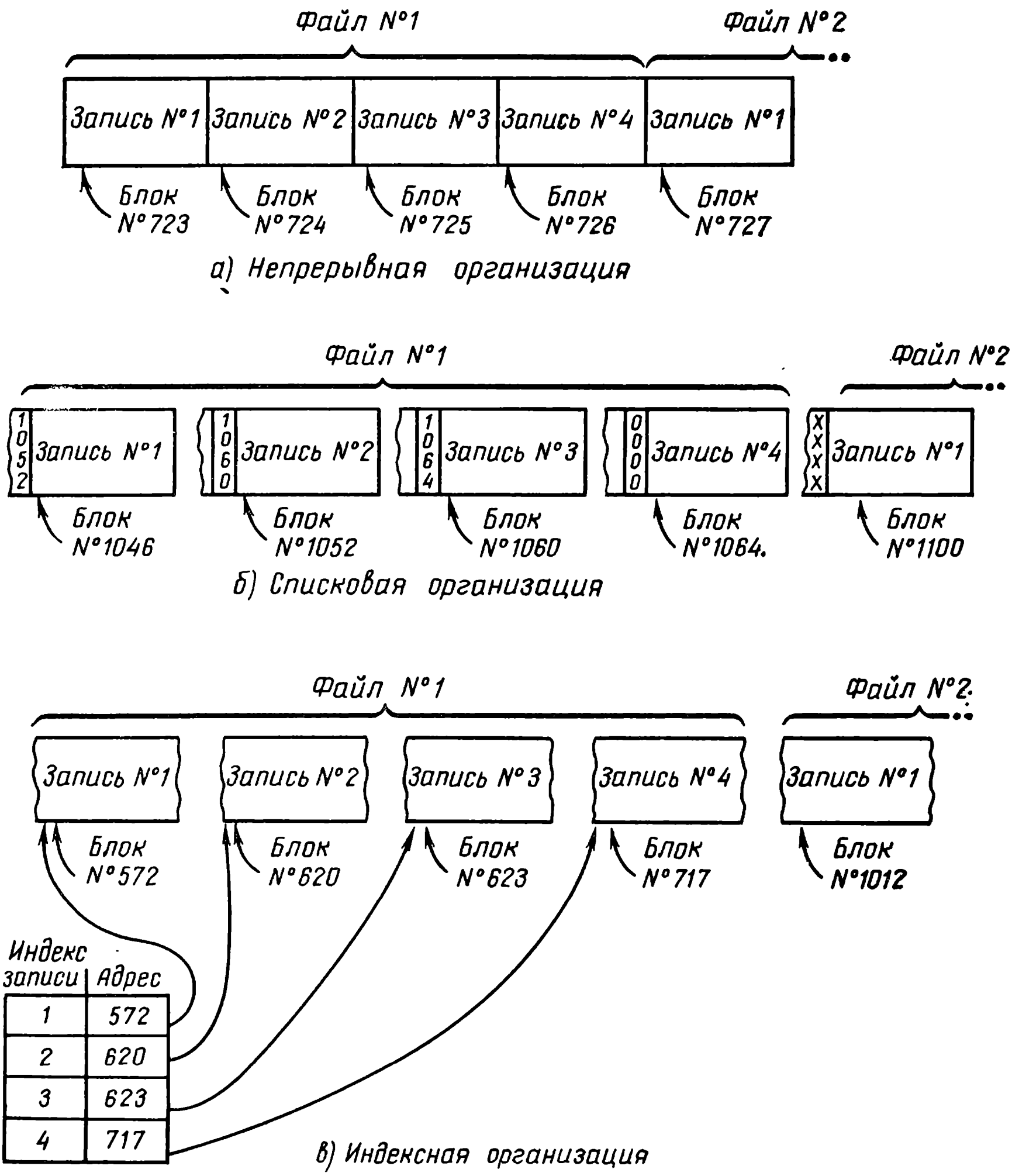


Рис. 10.1. Организация файлов

же и методы доступа определены системой, они непосредственно связаны с файловой структурой. Наиболее распространенным является *последовательный метод* доступа и к самим файлам, и к отдельным записям в них. Он особенно характерен для устройств, не имеющих прямого доступа, таких, как магнитные ленты, хотя и на устройствах типа дисков, тоже можно организовать последовательный доступ.

Последовательный метод доступа — такой способ хранения информации, при котором поиск файлов и записей в файлах выполняется в той последовательности, в какой они расположены на физическом носителе. Если необходимо найти начало некоторого файла или запись внутри него, то время, затрачиваемое на поиск, определяется длительностью просмотра всех файлов или записей до тех пор, пока не будет найдена нужная информация.

Обычно на таких устройствах, как магнитная лента, для последовательных файлов резервируется непрерывная область памяти. После того как обработана очередная запись, головка чтения/записи устройства сохраняет свою позицию, поэтому имеется непосредственная возможность для доступа к следующей записи.

Альтернативой последовательной организации файлов является *списковая организация,* которая позволяет создавать файлы на устройствах с *прямым доступом,* например на магнитных дисках. При подобной организации время доступа мало зависит от положения записи в файле. Списковый файл обладает еще и тем преимуществом, что по мере его увеличения к списку просто добавляются очередные элементы. При последовательной организации файлов в общем случае это невозможно, поскольку соседний блок на носителе уже может быть занят.

Существует еще один метод доступа, называемый *произвольным.* При этом методе доступа к файлам или записям в файле время доступа не зависит от расположения данного файла или записи относительно других на носителе. Поэтому последовательность обращения к файлу не имеет значения и может быть произвольной.

Для реализации этого метода доступа имеются также две возможности: может быть использовано непрерывное или индексное резервирование памяти. Если известно, где начинается непрерывный файл, то произвольный доступ может быть организован так же, как к элементам одномерного массива. Ограничения непрерывного способа резервирования остаются прежними: в общем случае размер файла не может быть увеличен.

Использование индексного способа резервирования позволяет как организовать произвольный доступ, так и при необходимости увеличивать размеры файлов. Такое резервирование имеет преимущества во всем, за исключением времени доступа, поскольку в этом случае, как и при списковом резервировании, прежде чем получить доступ к записи, необходимо обнаружить ее местонахождение.

Существует третий, промежуточный, метод доступа. Он может быть использован при работе с дисками и подобными им устройствами и позволяет произвольный доступ к файлам, но внутри файлов возможен только последовательный доступ к записям. Этот метод называется *индексно-последовательным,* в нем индексно адресуемый блок содержит более чем одну запись.

Индексно-последовательная организация подходит для тех случаев, когда к набору записей нужно иметь произвольный доступ, но сами записи в наборе организованы последовательно. Типичным примером такого применения файловой структуры может служить файл с записями о персонале служащих в какой-нибудь организации; после того как обнаружены записи, относящиеся к определенному сотруднику, дальнейшая работа может выполняться последовательно.

10.2.2. Оглавления

Если уже существует некоторая файловая структура и имеются методы доступа, то возникает проблема — как адресоваться

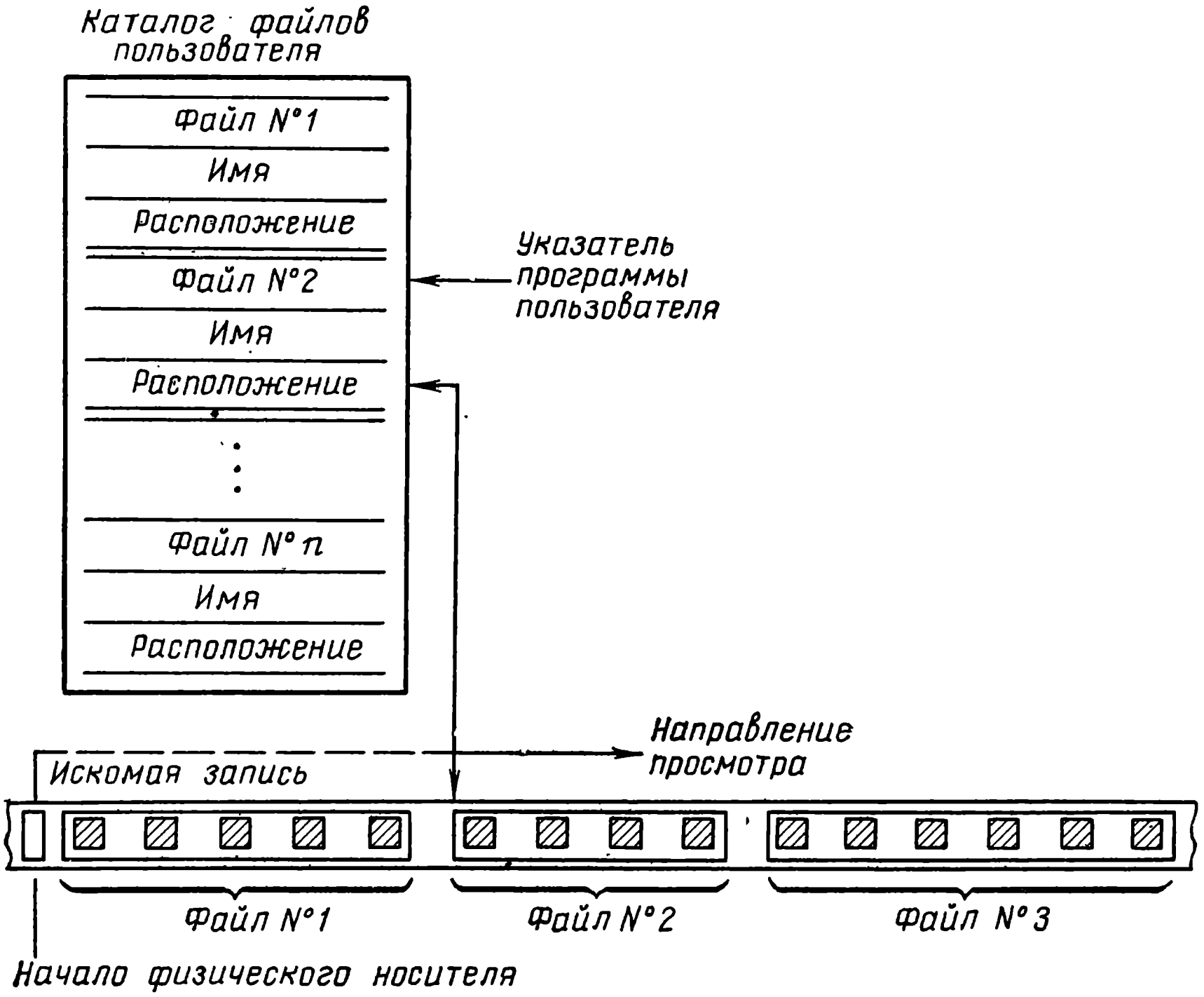


Рис. 10.2. Произвольный метод доступа к данным

к файлу, ранее запомненному в системе. Один из подходов состоит в том, чтобы использовать непосредственно физический адрес файла на устройстве. Однако это не очень удобно по тем же причинам, по каким при программировании физическим адресам предпочитают символические. Очевидно, что удобнее поставить в соответствие каждому файлу имя или идентификатор, и таким образом получить возможность доступа к файлам по именам.

Для обеспечения соответствия между именами файлов и их расположением на физических устройствах существует *оглавление файлов* — таблица, являющаяся частью системы, содержащая необходимую информацию о каждом файле. Оглавление может содержать не только имя файла и соответствующий ему начальный адрес на устройстве, но также информацию о файловой структуре, в том числе, если необходимо, и указатель в таблицу индексов. На рис. 10.2 показан произвольный метод доступа к последовательно организованным файлам посредством оглавления. Такой способ доступа может быть использован для работы с кассетной лентой или лентой типа DECtape, он позволяет знать адрес файла до начала работы с ним.

Если файлы, записанные в оглавлении, не хранятся в системе постоянно, а кассета или лента могут быть заменены, то собственно файл оглавления необходимо сохранять. Требуется, чтобы файл оглавления хранился на том же физическом носителе, на котором хранятся файлы, чьи имена в нем содержатся, причем его расположение должно быть определено заранее. Частью оглавления могут быть биты *отображения*, отмечающие, какие блоки устройства и какими файлами заняты, а какие — свободны.

Файловая структура, реализуемая с помощью оглавлений, обеспечивает простой и быстрый способ доступа к файлам. В этом состоит явное преимущество по сравнению с устройствами, на которых оглавления не могут быть использованы, например обычные магнитные ленты; доступ к файлам в этих устройствах осуществляется только последовательным просмотром.

10.2.3. Многоуровневые оглавления

В том случае, если несколько пользователей обращаются к одному и тому же устройству, например к системному диску, возникает проблема дублирования имен. Не зная, что какое-то имя файла уже задействовано, пользователь может повторить его и тем самым изменить или модифицировать чужой файл. Эта проблема решается введением для каждого пользователя отдельного оглавления, что позволяет ему назначать имена независимо от других пользователей.

Принципиально механизм пользования разделяемыми устройствами памяти требует двухуровнего файла оглавлений; это показано на рис. 10.3. Каждый пользователь имеет собственный код, который он сообщает системе при входе в нее, т. е. при выполнении команды LOGIN. Этот код служит для идентификации в главном файле оглавлений, который, в свою очередь, содержит ссылку на конкретное пользовательское оглавление.

10.2.4. Проблемы управления

Теперь, после обсуждения схемы работы с файлами, становится очевидным, что одной из центральных функций, выполняемых операционной системой, является *управление*. Это обусловлено

тем, что, например, файловая система как одно из мощных системных средств нуждается в управлении при резервировании памяти на внешних устройствах и при обмене информацией с ними. Отметим, что принципы построения самой файловой системы способствуют минимизации проблем при решении этой задачи.

В большинстве случаев собственно работа с устройствами не интересует пользователя, больший интерес для него представляют операции с наборами информации или файлами. Исходя из этого, целесообразно распространить файловый подход на все устройства: например, перфоввод можно интерпретировать как

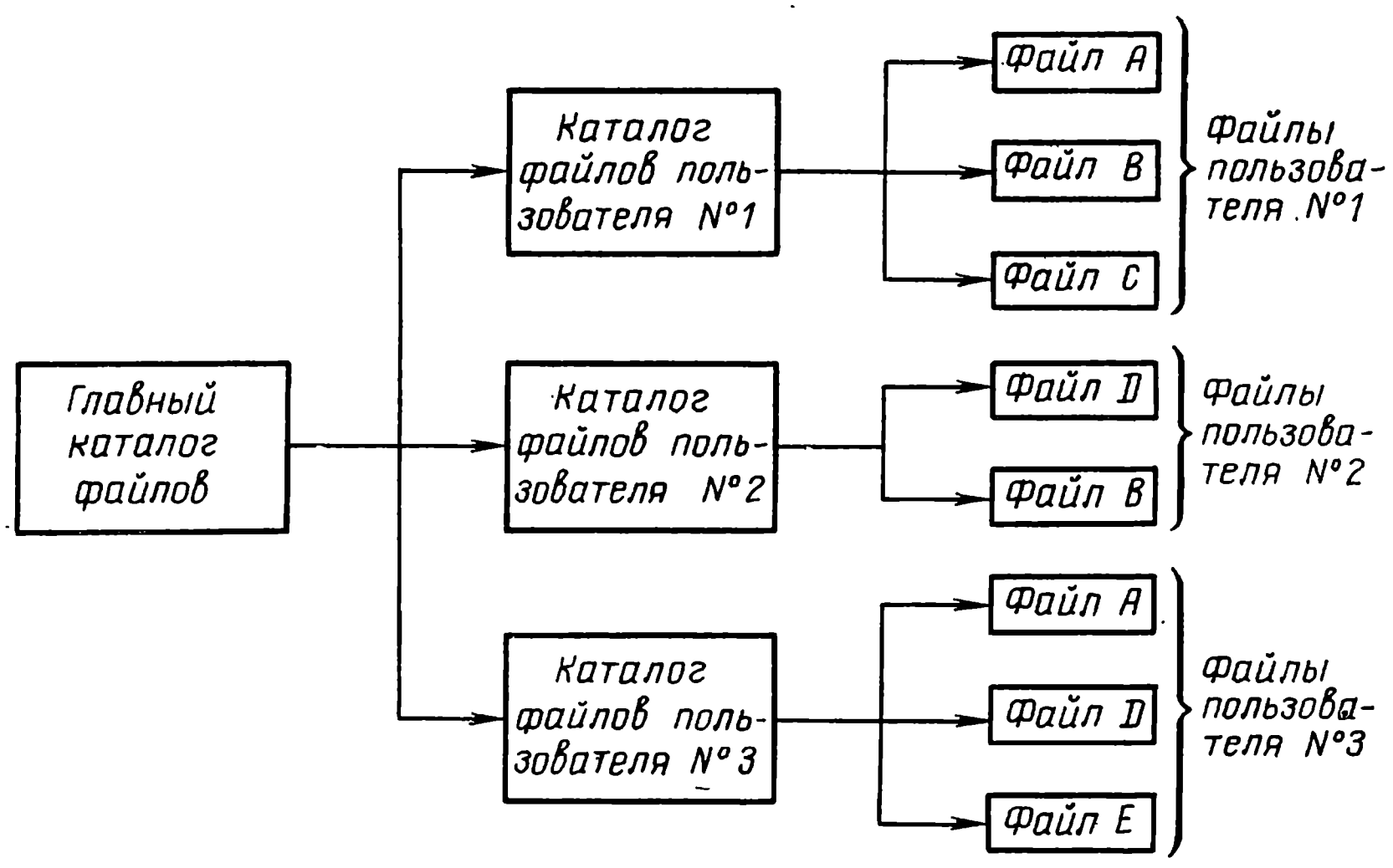


Рис. 10.3. Главный и пользовательский каталоги

устройство, работающее с файлами специального типа. Под *специальным типом* понимается то, что методы доступа к информации на этом устройстве имеют определенные ограничения. Введя их, можно любые устройства в системе интерпретировать как файловые. Для этого нужно только допустить, что не все файлы могут быть прочитаны, записаны или использованы в качестве исполняемой программы.

И система, и пользователь могут единообразно работать с устройствами, однако система должна обеспечивать интерфейс между системными или пользовательскими программами и внешними устройствами. Задачи, решаемые средствами такого интерфейса с точки зрения минимизации программирования, описаны в гл. 7. Было показано, что использование служебных программ (IOPS) освобождает пользователя операционной системы RT-11 от бремени программирования операций ввода-вывода, манипулирования файлами, организации одновременной работы различных устройств I/O, а также излишней зависимости от определенного типа устройств. Последнее положение особенно важно для дисковых операционных систем, так как в них программы обслужива-

ния устройств, не имеющих файловой структуры, например считыватель с перфоленты, не должны воспринимать как ошибочные обращенные к ним команды поиска файла, которые требуются для получения данных с устройств, имеющих файловую организацию.

Итак, основная функция IOPS состоит в организации информационного взаимодействия между системой и устройствами. Эта функция должна выполняться независимо от пользователя, поскольку одна из первоначальных целей IOPS заключалась в минимизации процедур, выполняемых пользователем, при работе с системой. Исходя из этого, центральной частью IOPS должны стать служебные программы для выполнения операций I/O, способные выполнять следующие функции:

1) управление устройствами;
2) работа с файлами на устройствах;
3) распределение памяти на устройствах;
4) ведение записей о запросах пользователей и текущем состоянии устройств;
5) координация действий при работе с периферийными устройствами для выполнения операций I/O, таких, например, как буферизация и организация блоков информации.

В гл. 8 было показано, что средством взаимодействия пользовательских программ со служебными программами устройств являются системные директивы; их также можно назвать *программными запросами.* В табл. 10.2 показаны основные директивы системы RT-11. Не все из них служат для непосредственного выполнения операций I/O. Одни, например .ENTER и .CLOSE, необходимы для инициализации набора данных или файла для последующих операций I/O, другие, например .DSTATUS и .FETCH, обеспечивают выполнение дополнительных операций с данными, файлами или устройствами.

Таблица 10.2. **Основные системные директивы**

.CLOSE	Прекращает работу с каналом, указанным в директиве
.DELETE	Удаляет указанный файл на заданном устройстве
.DSTATUS	Передает информацию об указанном в директиве устройстве
.ENTER	Резервирует на устройстве пространство, необходимое для файла
.EXIT	Прекращает выполнение пользовательской программы и передает управление в монитор
.FETCH	Загружает в память указанную в директиве служебную программу устройства
.GTIM	Позволяет программе пользователя получить текущее время и дату
.READ/.WRITE	Позволяет программе пользователя записывать или читать файлы с системного устройства
.SPND/.RSUM	Позволяет пользователю временно приостановить, а затем возобновить исполнение программы
.TTYIN	Передает символы с терминала в программу пользователя
.TTYOUT	Передает символы из программы пользователя на терминал
.WAIT	Прерывает выполнение текущей программы до того времени, пока не будут закончены операции I/O по указанному в директиве каналу

10.2.5. Утилиты работы с файлами

Системные директивы обеспечивают выполнение основных функций над файлами, однако при работе с данными часто приходится выполнять одни и те же операции и было бы неудобно каждый раз писать для этих целей новые программы. В составе системного программного обеспечения большинства операционных систем имеются специальные программы для обмена файлами между различными устройствами, эти же программы могут выполнять некоторые простые функции редактирования и управления. В операционной системе RT-11 используется программа обмена с внешними устройствами — PIP, позволяющая работать с любыми форматами данных и любыми файлами, допустимыми в этой операционной системе. Программа PIP выполняет следующие функции:

1) передает файл или группу файлов с одного устройства на другое;

2) сливает несколько файлов в один;

3) удаляет, модифицирует, переименовывает и перемещает файлы;

4) резервирует место для файлов и инициализирует устройства;

5) распечатывает листинги оглавлений;

6) выполняет защиту файлов.

На пользовательском уровне программы-утилиты выполняют служебную функцию, примерно соответствующую функциям системных директив на программном уровне. Пользователю нужно только ввести команду PIP, она будет декодирована, функции, указанные в ней, будут выполнены. Например, пользователь хочет сделать на флоппи-диске копию файла, существующего на диске. Для этого ему нужно запустить программу PIP, а затем, когда на терминале появится ее запрос, индицируемый звездочкой, напечатать следующее сообщение:

```
DX1:BACKUP.FOR=DK0:MYFILE.FOR
```

В соответствии с ним будет сделана копия BACKUP.FOR исходного файла MYFILE.FOR. В более поздних версиях RT-11, применяющих стандартный язык команд фирмы DEC — DCL, используется несколько измененная форма команды:

```
COPY DK0:MYFILE.FOR DX1:BACKUP.FOR
```

Для анализа состояния оглавления некоторого устройства может служить следующая команда:

```
LP:=DK0:/L (or DIR DK0: using DCL)
```

которая позволяет напечатать листинг *оглавления* нулевого диска на АЦПУ. Этот листинг имеет такой вид:

```
 1-JUL-78
MONITR.SYS    58 15-MAY-78
TT    .SYS     2 15-MAY-78
DX    .SYS     2 15-MAY-78
PIP   .SAV    14 15-MAY-78
LINK  .SAV    25 15-MAY-78
MACRO .SAV    31 15-MAY-78
FORTRA.SAV    96 15-MAY-78
SYSLIB.OBJ    37 15-MAY-78
RUNOFF.SAV    30 26-JUN-78
TECO  .SAV    27 15-MAY-78
VT52  .TEC     5 15-MAY-78
VEG            4 15-MAY-78
EDIT  .FIL     1  1-JUL-78
EDIT  .SAV    19 15-MAY-78
SYSMAC.SML    20 16-MAY-78
EDIT  .BAK     1 30-JUN-78
16 FILES, 372 BLOCKS
 108 FREE BLOCKS
```

Поля листинга используются для указания имени файла, его расширения, размеров и даты создания. *Расширение имени файла* служит для обозначения формата файла или методов его применения. Так, файлы с расширением «SYS» содержат машинные коды и используются системой; файлы с расширением «SAV» — программы в машинных кодах, которые могут быть загружены и выполнены; расширение «OBJ» обозначает файлы, полученные в результате работы ассемблера, в форме подготовленной для компоновщика или связывающего загрузчика.

Утилиты работы с файлами выполняют такую же функцию, как программы в гл. 8. Назначение их состоит в том, чтобы помочь пользователю при выполнении некоторых операций. Сами по себе утилиты не порождают новых и полезных результатов, но служат для повышения эффективности работы пользователя.

10.2.6. Независимость от типов устройств

Системные макрокоманды ввода-вывода порождают ловушки для работы по прерываниям и, следовательно, при этом должны указываться устройства, обмен с которыми необходим. Однако могут возникнуть обстоятельства, когда пользователю может потребоваться переназначение устройства перед исполнением программы. Например:

1) устройство, заданное при написании программы, недоступно, но можно заменить его другим, аналогичным;

2) программист заранее не знает конфигурацию системы, на которой будет работать его программа или не может ориентироваться на нее (допустим, что создается пакет общего назначения).

Использование таблицы устройств (внутри системы RT-11 она называется UNT) и команды ASSIGN позволяет писать программы, которые не зависят от типов устройств. Команда ASSIGN

назначает определенное программистом логическое имя определенному физическому устройству. С точки зрения пользователя, независимость от типов устройств увеличивает гибкость программирования.

10.2.7. Монитор

Взаимодействие пользователя с системой осуществляется двумя способами: первый — выполнение команд с пульта терминала, второй — ответы на запросы программ. В обоих случаях инициируется некоторая *управляющая программа,* выполняющая загрузку файла, устанавливающая соответствие между логическим файлом или физическим устройством, открывающая файл, выводящая запись на устройство и т. д.

Для успешной работы системы различные управляющие программы должны совместно работать в определенном взаимодействии. Во многих системах *задачи* выполняются последовательно одна за другой, однако часто несколько их работают параллельно, например при вычислениях и операциях с внешними устройствами. Здесь под задачей понимается некая работа, которая определена с достаточной полнотой для того, чтобы участвовать в конкурентной борьбе за ресурсы системы — память, внешние устройства и т. д. Очевидно, что необходима главная управляющая программа — *монитор,* которая ответственна за функционирование всей системы.

Монитор несет ответственность за инициацию, выполнение и окончание работы каждой программы. Он координирует взаимодействие программ и задач между собой, связи между пользователями, работу системы и отдельных управляющих программ. Кроме того, он должен реагировать на вызовы других программ, фиксировать и передавать обращения I/O к программам обслуживания устройств, следить за работой с данными и файлами и обеспечивать диагностику ошибок.

Обычно монитор состоит из трех основных частей: резидент, нерезидентная часть и загрузчик системы. Резидент монитора постоянно находится в памяти, выполняя роль интерфейса между программой и системными ресурсами

При исполнении пользователем команд нерезидентной части может быть изменена структура резидента. Наличие нерезидентной части позволяет пользователю добавлять к системе модули, необходимые для исполнения очередной задачи. Обычно по окончании работы задачи управление передается в нерезидентную часть монитора. В этот момент пользователь имеет возможность выполнить нужную последовательность команд для настройки системы на работу со следующей задачей.

Загрузчик системы служит для построения новой версии резидента в соответствии с предварительно выполненными командами нерезидентной части. Он загружает всю систему и заголовки тех программ, которые дают пользователю возможность редактировать, ассемблировать, загружать, выполнять и отлаживать

программы и осуществлять работу с файлами. Работа системного загрузчика незаметна для пользователя, потому что в основном она состоит в настройке системы путем загрузки и установки отдельных системных программ.

Взаимодействие пользователя с монитором. Основным интерфейсом пользователя с операционной системой RT-11 является терминал. Для управления используются команды монитора, которые позволяют загружать и выполнять системные и пользовательские программы (см. разд. 3.2), а также некоторые команды, обеспечивающие выполнение специальных функций; для управления программами применяются также служебные символы.

Большая часть команд монитора должна быть введена до загрузки той программы, к исполнению которой подготавливается система. Команды интерпретируются модулями, входящими в состав нерезидентной части монитора, поскольку во время работы пользовательской или системной программы нет нужды хранить в оперативной памяти программы, которые распознают и интерпретируют все команды монитора. Однако небольшое подмножество команд должно быть доступно во время работы программ. Эти команды интерпретируются тем разделом драйвера терминала, который входит в состав резидентной части монитора; они позволяют запускать программу (в том числе и повторно), распечатывать содержимое памяти и перезагружать нерезидентную часть монитора.

В связи с тем, что и работающая программа, и монитор получают информацию с одного и того же терминала, пользователь должен указывать, куда он передает свое сообщение. Для указания режима работы используются специальные символы. Все символы, следующие за специальным (в RT-11 это CTRL/C), после прерывания программы передаются монитору. Остальные символы, печатаемые на терминале, адресуются работающей программе; они поступают в буфер и хранятся в нем до тех пор, пока не будут затребованы ею.

Организация монитора. На рис. 10.4 показаны общая организация монитора и диаграмма потока данных. Хотя функции большинства модулей, показанных на этом рисунке, уже обсуждались выше, для некоторых требуются дополнительные пояснения. В первую очередь это касается модулей *процессора команд* и *декодера команд монитора*. Эти модули выполняют предварительную обработку последовательностей символов при вызове их либо из пользовательской, либо из системной программы. Для обеспечения работы универсальных модулей и пользовательские, и системные программы должны пользоваться общими форматами представления данных.

Следующий модуль, представляющий определенный интерес,— *таблица назначения устройств* (DAT). Она служит для хранения данных, введенных по каждой команде ASSIGN. Необходимость в такой таблице определяется тем, что описание соответствия между файлом и устройством с терминала может быть выполнено

в произвольный момент времени, даже тогда, когда программа, использующая соответствующее назначение, уже загружена в память. Таблица DAT размещается внутри монитора, поэтому обращение к ней может быть выполнено, когда работающая программа начинает обмен данными.

Следующий пример демонстрирует применение таблицы назначения. Допустим, что прежде, чем запустить некоторую пользовательскую программу, нужно заменить обращение к флоппи-диску на обращение к магнитной ленте типа DECtape. В таком случае должна быть выполнена следующая команда:

```
ASSIGN DX1:DT0
```

Здесь двоеточие разделяет физическое имя устройства (для флоппи-дисков это DX1) от нового логического назначения (для ленты типа DECtape это DT0).

На первый взгляд использование команды ASSIGN и таблицы назначений устройств может показаться странным. Однако программистам, знакомым с Фортраном, станет ясно, что таким образом решается по-новому одна старая проблема. При программировании на Фортране для выполнения чтения или записи следует использовать операторы такого типа:

```
READ  (u,f) I/O list
WRITE (u,f) I/O list
```

где u — номер устройства и f — метка оператора FORMAT. По умолчанию обычно для устройств чтения с перфокарт используется номер 5 и оператор чтения с перфокарт выглядит так: READ (5, 10) A, B, C, а для алфавитно-цифрового печатающего устройства — номер 6 и оператор печати может быть таким: WRITE (6, 20) A, B, C. Если возникает необходимость вместо этих устройств применить какое-либо другое, имеющее файловую структуру, например магнитную ленту, то необходимо выполнить переназначение. Новое назначение будет относиться ко всем файлам на устройстве, которому присвоен соответствующий номер; пользователь сам должен различать отдельные файлы.

Таблица размещения монитора. Важной частью программы обработки прерываний является таблица размещения монитора (MTR), которая обеспечивает информацией программу обработки прерываний:

1. Показывает, где расположены программы монитора, находящиеся в памяти постоянно, и куда загружены остальные его компоненты на время выполнения.

2. Служит справочником, указывая, где расположены нерезидентные модули монитора на системном устройстве; тем самым обеспечивается необходимый доступ к ним, когда эти программы должны быть прочитаны в память.

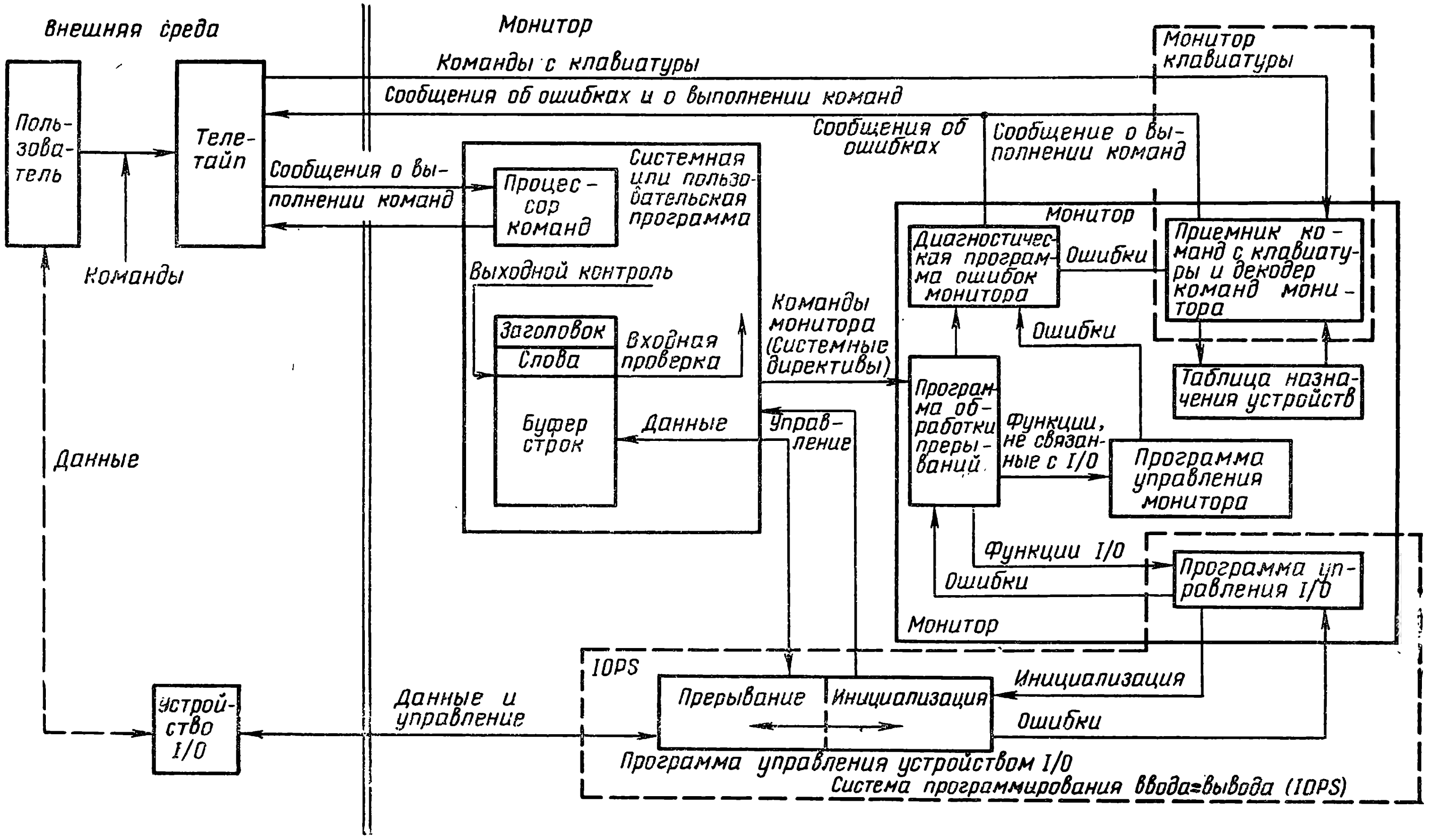


Рис. 10.4. Блок-схема монитора

Состояние таблицы определяется тем, какие программы должны размещаться в оперативной памяти постоянно, чтобы осуществлять общее управление работой системы, и тем, какие из них должны размещаться на системном устройстве, поскольку они используются лишь время от времени. Нерезидентные программы загружаются в память на время работы системным загрузчиком, а по окончании удаляются. Таким образом удается ограничить пространство памяти, необходимое для работы монитора, увеличивая пространство памяти, остающееся для пользователя; очевидно, что такое решение совершенно обосновано в связи с ограниченностью размеров оперативной памяти мини-ЭВМ.

Распределение памяти. Из сказанного ясно, что часть модулей монитора должна постоянно присутствовать в памяти. Они определяют необходимый минимум использования памяти (рис. 10.5). Модульность структуры монитора позволяет пользователю самому определить, какие модули должны составлять резидентную часть, а какие при необходимости должны загружаться с диска. Для загрузки нерезидентных программ в память резервируется область обслуживающих программ, достаточная для размещения в ней любой из них.

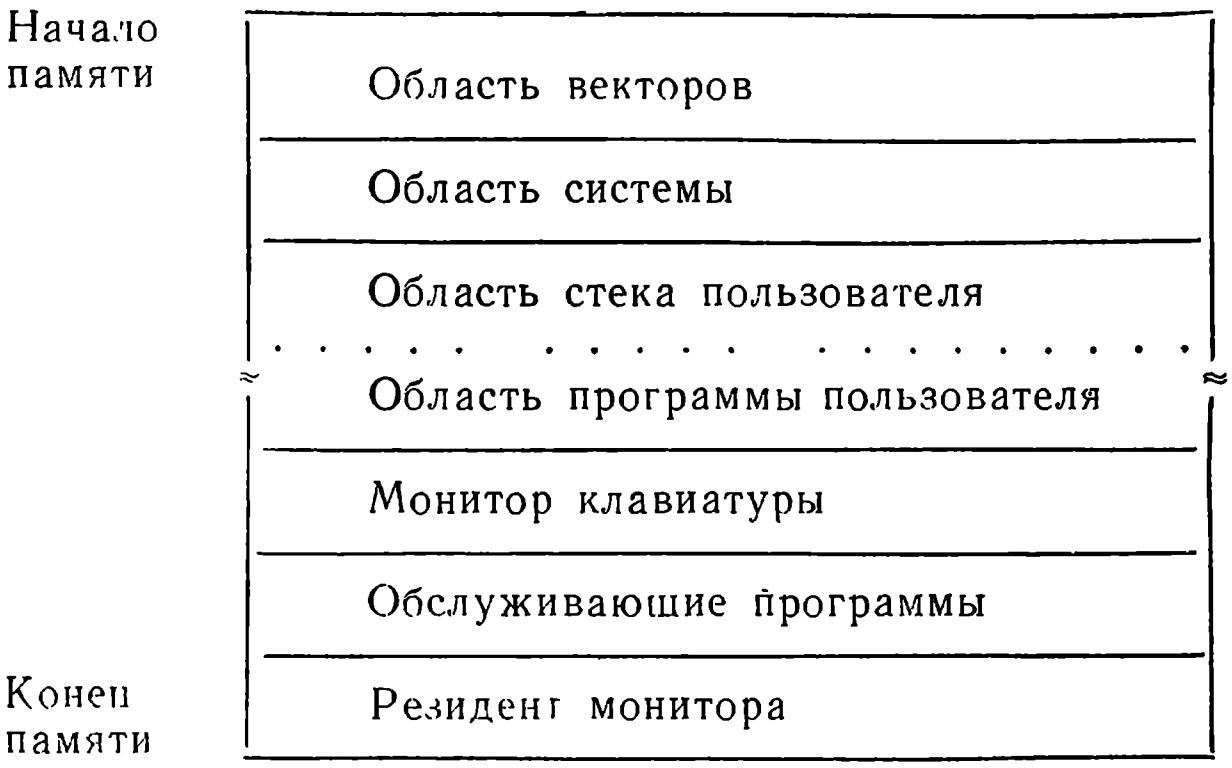


Рис 10.5. Распределение памяти

Несмотря на то, что обмен с диском выполняется достаточно быстро, он все же занимает определенное время; поэтому в тех случаях, когда пользователь имеет достаточную память и возникает необходимость в ускорении работы системы, можно, используя свойство модульности операционной системы, поступить двояко:

1) если некоторый модуль используется достаточно часто, он должен быть включен в список модулей, составляющих резидентную часть монитора;

2) если модуль часто применяется совместно с какой-либо программой, его можно загружать одновременно с ней; таким образом, для нее этот модуль будет резидентным.

Динамическое распределение памяти. Схема распределения памяти с использованием динамического буфера (см. рис. 10.4) представляет собой еще одно средство монитора. Эта схема позволяет отложить резервирование памяти для операций ввода-вывода до тех пор, пока в работающей программе не возникает реальная потребность в них. Только в этот момент выполняется загрузка драйверов, и после завершения процедур обмена они удаляются из памяти. Под динамическим распределением памяти понимается свойство основной структуры распределения памяти изменяться во времени.

Следствием этого свойства памяти является то, что соответствующие модули должны допускать загрузку в произвольное место памяти. Создание позиционно-независимых программ для такой машины, как PDP-11, не представляет проблемы, но существует проблема организации их взаимодействия. Для решения этой проблемы может служить *таблица векторов системы* (SVT), которая представляет собой общую область, содержащую информацию о состоянии системы в любой момент времени. SVT включает, в частности, указатели на различные части операционной системы, например адрес последней ячейки монитора, начало таблицы размещения монитора, имя загруженной программы и ряд других параметров.

10.3. ИСПОЛЬЗОВАНИЕ ОПЕРАЦИОННЫХ СИСТЕМ

После того как была обсуждена внутренняя структура типичной дисковой операционной системы, рассмотрим, каким образом операционная система удовлетворяет потребности пользователей. Очевидно, что эти потребности очень разнообразны, и столь же очевидно, что нельзя создать операционную систему, полностью удовлетворяющую все потребности пользователей в конкретных прикладных областях. Для разрешения этого противоречия разработчики предлагают в каждом конкретном случае выполнять генерацию операционной системы. Области применения и условия могут быть классифицированы по-разному и в том числе так:

1. Системы, работающие в пакетном режиме или в режиме разделения времени.
2. Системы управления, работающие в режиме реального времени.
3. Системы баз данных.
4. Коммуникационные вычислительные системы.

Системы, работающие в пакетном режиме или в режиме разделения времени, знакомы большинству пользователей. Это системы общего назначения, в наилучшей степени приспособленные для разработки и выполнения программ. Для мини-ЭВМ они представляют собой некоторую надстройку к основным дисковым операционным системам.

Системы управления, работающие в режиме реального времени, проектируются для условий, когда необходимо поддерживать

выполнение и управлять большим числом задач с учетом событий, которые происходят вне вычислительной машины. Такие системы должны поддерживать расписание работы программ реального времени, реализуя необходимые функции, связанные с вводом и выводом, оповещением обо всем происходящем человека-оператора и ряд других функций, требуемых для мультипрограммной системы, работающей в режиме реального времени. Обычно такие системы служат для управления технологическими процессами. В этих случаях система должна быстро реагировать на происходящие асинхронные внешние события (например, срабатывание выключателя сигнала тревоги в устройстве, управляемом системой), и управление должно быть как можно быстрее передано соответствующей обслуживающей программе.

Системы баз данных отличаются тем, что они оперируют огромными объемами информации, которая должна быть доступна для пользователя, выдавшего запрос, и в то же время защищена от случайных потерь и искажений при выполнении недопустимых команд. Как и в системах реального времени, здесь основное внимание уделяется не разработке или отладке программ, а их использованию.

Коммуникационные вычислительные системы обычно находят применение в промышленных распределенных системах, например при снабжении электроэнергией или природным газом. И в том и в другом случае сама вычислительная система представляет собой распределенную сеть взаимосвязанных устройств, которая допускает неограниченный рост по мере возрастания потребностей пользователя. Система может расширяться за счет увеличения числа устройств и за счет расширения функций существующих элементов. Такие комплексы требуют точного определения интерфейсов и структур взаимодействия.

Далее в этой главе будут рассмотрены особенности каждой из описанных выше операционных систем. Поскольку предметом данной книги являются мини-ЭВМ, основное внимание будет уделено операционным системам, которые были разработаны для таких ЭВМ, и тем функциям, которые они выполняют.

10.4. СИСТЕМЫ, РАБОТАЮЩИЕ В ПАКЕТНОМ РЕЖИМЕ И РЕЖИМЕ РАЗДЕЛЕНИЯ ВРЕМЕНИ

Построение *системы, работающей в пакетном режиме*, на основе дисковой операционной системы в сочетании с возможностями мини-ЭВМ не представляет значительной сложности. Процессор, управляющий пакетным режимом, представляет собой дополнительную по отношению к монитору программу, которая делает возможным ввод команд и программ, выполнение которых затребовано этими командами, с одного и того же устройства. Набор таких команд и программ, для которых четко определен порядок выполнения, образует единицу, называемую *заданием*.

Система допускает возможность последовательного выполнения большого количества заданий без вмешательства оператора.

Помимо основных команд монитора, для организации пакетного режима используются следующие команды:

1. Войти в пакетный режим.
2. Определить и разделить задания.
3. Задать поток данных.
4. Указать конец задания.
5. Выйти из пакетного режима.

Совокупность этих команд образует *язык управления заданиями* и в случае, когда в качестве устройства ввода используется считыватель с перфокарт, команды перфорируются на управляющих картах.

Кроме пакетных систем, на мини-ЭВМ часто находят применение *системы с разделением времени.* В общем систему с разделением времени можно определить как систему, в которой несколько пользователей одновременно имеют доступ к центральным вычислительным средствам.

Такой комплекс, по сути дела, представляет собой *мультипрограммную* ЭВМ, в которой ресурсы распределены между различными пользователями таким образом, что у каждого из них создается впечатление монопольного использования машины. Если система мультипрограммна, то в один и тот же момент несколько пользовательских программ размещается в оперативной памяти. Каждая из них получает свою порцию времени CPU, называемую *квантом времени*, в течение которой она может выполнять требуемые вычисления. Когда квант времени закончился или программа достигла точки, в которой дальнейшие вычисления невозможны (например, необходимо получить некоторую информацию с устройства I/O), ресурсы CPU передаются другой программе. Такая передача управления не занимает много времени, потому что следующая выполняемая программа уже находится в памяти.

Работа систем с разделением времени основана на временном динамическом разделении времени CPU и пространства памяти. В некоторых случаях все программы пользователей могут быть резидентными, но чаще постоянно размещена в памяти только часть из них, а остальные загружаются с внешних устройств. В тех случаях, когда размер памяти недостаточен для хранения всех пользовательских программ и данных, приходится организовывать обмен с такими устройствами.

По сравнению с дисковой операционной системой система с разделением времени (TSOS) имеет целый ряд дополнительных средств, в том числе сложные программы управления распределением системных ресурсов, квантования времени, распределения памяти, перемещения программ, а также основные сервисные средства пользователя. Одна из таких управляющих программ — *планировщик* — отвечает за обслуживание пользователей и оптимальное использование системных ресурсов. Каждый раз, когда монитор получает управление, он вызывает планировщика, чтобы

определить, какая программа должна быть выполнена следующей и какие обмены с внешними устройствами следует выполнить для обеспечения занятости системы и запросов пользователей.

Гибкость систем с разделением времени часто ограничивается возможностями ЭВМ. Самыми гибкими, дающими пользователю прямой доступ ко всем ресурсам операционной системы, в том числе и к устройствам ввода-вывода, являются системы с открытой *формой взаимодействия пользователя с системой. Системы с закрытой формой взаимодействия* ограничивают применение системных средств и включают только определенные языки программирования. Наиболее ограниченным типом операционной системы с разделением времени по предоставляемым пользователю возможностям являются *системы дистанционного ввода заданий.* В них пользователь имеет возможность подготовки программ на удаленном терминале и затем передачи их на выполнение в систему, однако он не может организовать взаимодействие с программой во время ее работы, ему доступно лишь получение окончательных результатов по окончании работы программы.

Из операционных систем с разделением времени на мини-ЭВМ наибольшее распространение имеют системы с закрытой формой взаимодействия. Обычно они дают возможность 8—64 пользователям работать одновременно на языках высокого уровня, имея доступ к большей части системных ресурсов. Обычно число пользователей в системе ограничивается не производительностью процессора, а размерами оперативной памяти.

10.5. СИСТЕМЫ УПРАВЛЕНИЯ, РАБОТАЮЩИЕ В РЕЖИМЕ РЕАЛЬНОГО ВРЕМЕНИ

Системы реального времени создаются в целях обработки данных в темпе реального процесса, обеспечивающего систему требуемой информацией. Подобные системы обычно работают в мультипрограммном режиме, их мониторы в режиме реального времени управляют большим числом задач, резидентных в оперативной или в дисковой памяти. Это управление обеспечивает разделение ресурсов между задачами, и в том числе распределяются оперативная и дисковая память, программы управления внешними устройствами.

Дисциплина выполнения большого числа задач определяется их приоритетами, аппаратными прерываниями, алгоритмами распределения времени и запросами от других задач. В системе реального времени пользователь имеет возможность установить новые задачи, работающие в режиме реального времени, присвоить им приоритеты и задать периодичность их выполнения с произвольным временным интервалом

Время реакции системы на запрос на выполнение определенной задачи в основном определяется приоритетом последней. Для этого применяется *специальный таймер,* который не позволяет программам с высоким приоритетом работать слишком долго, и

гарантирует возможность обслуживания всех задач. В соответствии с приоритетом задачи перед выполнением в таймер засылается значение максимального интервала времени, в течение которого она может непрерывно работать до остановки или снятия с исполнения.

Монитор реального времени управляет всеми операциями ввода и вывода. Выполнение этой функции очень важно для пользователей системы реального времени в связи с тем, что функционирование такой системы характеризуется большим числом операций ввода-вывода. В принципе выполнение задач в ней инициализируется или прерывается по выполнении различных внешних операций.

10.5.1. Программирование в системе реального времени

При программировании в системах реального времени в основном используют язык Ассемблера или Фортран с расширением для вызова монитора. Разработка программ может выполняться под управлением монитора реального времени, но ограниченния по памяти или сложность системы могут потребовать использования для разработки другой операционной системы.

Для работы в составе системы программы реального времени при организации последовательности работы, очередей, синхронизации и т. д. применяют системные макрокоманды. Часто данные, которые собирает задача, не обрабатываются сразу, а просто запоминаются для последующей обработки в операционной системе общего назначения.

10.6. СИСТЕМЫ БАЗ ДАННЫХ

Система *базы данных* как средство работы с текстовой информацией дает возможность доступа к общим данным большому числу пользователей. Она представляет собой координированный набор данных, например файлов. Примерами таких систем могут служить системы регистрации медицинской информации, резервирования билетов, информационно-справочные и автоматические библиографические системы.

Условия, в которых работают системы баз данных, характеризуются небольшими вычислительными потребностями, но значительными требованиями к возможностям памяти. При поступлении данных в систему они проверяются на допустимость, решается вопрос о том, куда их следует поместить, и формируется приемлемый ответ на возможный запрос пользователя. Ни одно из этих действий не требует значительных вычислений, они сводятся в основном к операциям над символами.

При подготовке отчета по запросу пользователя производится определенная обработка информации и может осуществляться доступ к внешним запоминающим устройствам. Эти операции также не требуют больших вычислительных работ, нужно только сформировать и воспроизвести отчет. В связи с этим система и

при центральном процессоре малой мощности может обслуживать большое число пользователей.

Хотя в системах баз данных должен быть обеспечен потенциальный доступ к большей части хранящихся в ней данных в любой момент времени, потребность в обращении к большим объемам информации может быть ограниченной. Таким образом, важным аспектом систем такого рода является возможность доступа к периферийным запоминающим устройствам — дискам или барабанам. Далее, для результативной работы система должна эффективно использовать пространство памяти, минимизируя число задействованных ячеек, обеспечивая быстрый и оперативный поиск данных.

10.6.1. Эффективная работа с данными

Эффективная работа с данными обеспечивается при соблюдении следующих условий:

1. Хранимые данные следует представлять в форме иерархической древовидной структуры так, чтобы те из них, к которым чаще всего обращаются, занимали оптимальное положение.

2. Восстанавливать и модернизировать информацию нужно одновременно.

3. Разные пользователи должны иметь различный доступ к данным. Например, при обращении к списку сотрудников фирмы одним пользователям необходимы имена и занимаемые сотрудниками должности, а другим — табельные номера и оклады.

4. Необходимо обеспечить возможность динамического изменения структуры, но это не должно быть разрешено пользователю.

5. Резервирование памяти следует выполнять не статически, а в зависимости от текущих требований.

6. Структура данных должна быть оптимально отображена на внешних устройствах памяти и переписываться или восстанавливаться только в случае необходимости.

7. Во избежание перепрограммирования необходимо делать систему независимой от конкретного типа устройств.

8. Работу с системой следует осуществлять с применением реентерабельных методов для того, чтобы различным пользователям можно было пользоваться одной копией программ и данных.

9. С целью минимизации числа обменов с внешними устройствами нужно стремиться максимизировать количество системных и пользовательских задач, резидентных в оперативной памяти.

При разработке программ информационной системы одним из наиболее важных аспектов эффективности использования машинного времени является оптимальное взаимодействие с системой в конкретной прикладной области. Необходимо также уделять большое внимание инженерной психологии и возможности изменения и пересмотра методов, доступных пользователю, для хранения и восстановления системных данных. Наконец, требуются гарантии защиты и секретности информации.

10.6.2. Методы хранения данных, доступа и работы с ними

Сложность, связность, секретность и разнообразие данных, с которыми приходится работать пользователям, определяют ряд требований к самой системе. Так, поскольку значительная часть входной информации представляется в виде символьных строк переменной длины, система должна не только проверять синтаксис этих входных предложений, но часто и (если это возможно) анализировать их семантику на соответствие определенным ограничениям.

Одной из важнейших функций систем управления базами данных является предохранение данных от несанкционированного доступа. Для этого с каждым набором данных связан соответствующий список параметров, содержимое которого при каждом обращении к базе данных сравнивается с признаками доступа пользователя. Обеспечиваемые при этом секретность и право собственности на информацию селективны и зависят от самой информации.

Для защиты собственно системы используются различные способы, и в том числе реализация ее как закрытой системы с разделением времени. В этом случае пользователи могут только отвечать на предварительно подготовленные запросы, но не могут писать, тестировать и отлаживать программы общего назначения. Кроме того, возможно использование терминалов с заранее запрограммированной клавиатурой, где каждой клавише соответствует определенное хранящееся в системе сообщение. Целесообразно также применение специальных языков высокого уровня, с помощью которых реализуются более сложные логические цепочки доступа к данным или обеспечивается доступ к более сложным структурам данных. Построение таких языков на базе компиляционных, а не интерпретационных методов приводит к повышению степени защищенности системы.

10.7. КОММУНИКАЦИОННЫЕ ВЫЧИСЛИТЕЛЬНЫЕ СИСТЕМЫ

Коммуникационная вычислительная система функционирует как сеть взаимосвязанных вычислительных элементов, основанная на их взаимодействии и разделении ресурсов. В такой системе мини-ЭВМ может быть либо коммуникационным процессором, либо концентратором данных.

В качестве коммуникационного процессора мини-ЭВМ может выполнять следующие функции:

1) пересылка сообщений по сети в режиме «запомнить и передать» (store and forward);
2) формирование и трансляция сообщений;
3) реализация связи абонентских ЭВМ с сетью;
4) управление дистанционным вводом заданий.

В качестве концентратора данных мини-ЭВМ реализует функции следующих элементов сети:

1) мультиплексора для большого числа низкоскоростных локальных терминалов, концентрирующего данные в одну среднескоростную линию для большой системы;

2) буфера сообщений, контроллера линий связи и средства для сборки сообщений из символов или, наоборот, их разборки на символы для низкоскоростных устройств.

В обоих случаях использование мини-ЭВМ позволяет заменить дорогостоящие и сложные аппаратные контроллеры на входе больших вычислительных систем. Поскольку мини-ЭВМ являются машинами общего назначения, имеют развитые средства программирования и обработки прерываний, они могут, будучи соответственно запрограммированы, выполнять процедуры:

1) управления маршрутизацией сообщений;

2) преобразования кодов и скоростей передачи;

3) управления линией и обработки ошибок;

4) сжатия данных и формирования сообщений;

5) автоматической идентификации терминалов и их характеристик;

6) снабжения сообщений датой и временем;

7) автоматической установки связи;

8) предварительного анализа сообщений перед передачей;

9) редактирования, табуляции и других служебных операций форматирования сообщений.

10.7.1. Программное обеспечение коммуникационных систем

Программное обеспечение, поставляемое фирмами-изготовителями, поступает в двух видах. Первые — законченные системы, или, как часто их называют, *системы под ключ*,— могут быть установлены и непосредственно сразу запущены в работу. *Модульные системы,* состоящие из аппаратных средств, включая ЭВМ и специальную коммуникационную аппаратуру, и специального прикладного программного обеспечения, в том числе драйверы и коммуникационные исполнительные программы, составляют вторую группу. Коммуникационная система DECnet фирмы DEC представляет собой пример системы второго типа.

Системы под ключ не требуют от пользователя написания каких-либо программ. Некоторые из таких систем даже снабжаются постоянной памятью, содержимое которой не может быть случайно разрушено, программы содержащиеся в них состоят из фиксированной последовательности команд.

В отличие от систем «под ключ» модульные системы могут явиться базой, на основе которой пользователь имеет возможность построить комплекс специального назначения, соответству-

ющий его нуждам. Программное обеспечение такой системы включает программы обработки прерываний, прикладные программы для терминалов, прикладные пакеты для управления и создания интерфейсов. Используя их, пользователь может приспособить систему к своим прикладным задачам с минимальным объемом аппаратных средств и программного обеспечения.

БИБЛИОГРАФИЯ

Операционным системам посвящено значительное количество хороших книг. Однако в большинстве из них, например в Watson R. W. (1970), Katzan H. (1973), Donovan J. J. (1972) и Organick E. I. (1972), основное внимание уделено возможностям в структуре некоторых конкретных операционных систем, например OS/360, Multics или XDS-940. Примечательным исключением являются книги Brinch Hansen (1973), Cohen L. J. (1970), Denning P. J. and Coffman U. O. (1973) и Shaw A. C. (1974). К сожалению, последние четыре из них являются в большой степени теоретическими и математическими и не могут быть полезны тем, кого интересуют практические приложения. Для получения общего представления читателю стоит внимательно изучить статью исследовательского характера Denning P. J. (1973). Наконец, в книге Haberman A. N. (1976) уделяется внимание деталям проектирования операционных систем; в работе Holt R. S (1978) исследуются аспекты параллельных программных структур.

Глава 11

СТРУКТУРА И ПРИНЦИПЫ ФУНКЦИОНИРОВАНИЯ МУЛЬТИПРОГРАММНЫХ ОПЕРАЦИОННЫХ СИСТЕМ

Вычислительная система, допускающая формирование заданий из программ, определенных пользователем, служит важнейшим элементом при проведении научных и лабораторных экспериментов, а также в управлении процессами реального времени, требующими непрерывного, длительного контроля различных переменных. В качестве примеров здесь можно привести системы слежения за поведением животных в лаборатории, наблюдения за движением соков у растений, а также передачи сообщений от центральной ЭВМ к множеству удаленных терминалов.

Несмотря на кажущееся разнообразие, все эти функции могут успешно выполняться системой малых ЭВМ. Благодаря низкой стоимости и широким возможностям мини-ЭВМ может быть запрограммирована для обеспечения планирования задач управления I/O, связи с оператором и других функций, необходимых для решения многих прикладных задач. Большинство фирм-изготовителей таких машин предусматривает разработку пакетов прикладных программ реального времени, которые легко приспосабливаются к требованиям пользователя.

Данная глава посвящена описанию обобщенной системы мультипрограммирования (MMS), с помощью которой становится возможным одновременное выполнение нескольких заданий на PDP-11. Система MMS очень проста, но эффективно выполняет свои функции и требует минимума памяти и минимального времени на собственные нужды. Отметим, что MMS представляет собой открытую систему и, следовательно, при определенном уровне понимания ее структуры и функционирования может быть легко расширена (см. разд. 11.5 и упражнения в конце главы).

11.1. ОБЩИЙ ОБЗОР MMS

Поскольку стоимость операционной системы может быть выражена как функция ресурсов (как требующихся, так и используемых) и времени прохождения задания пользователя (например, длительность выполнения системных программ), важнейшей целью такой системы является минимизация занимаемого ею объема памяти. Для MMS он составляет приблизительно 200 слов, что обусловлено ее простотой. С другой стороны, это приводит к тому, что каждый пользователь должен разрабатывать собственные программы ввода-вывода. Кроме того, уменьшение тре-

буемого для MMS объема памяти достигается за счет отсутствия в ней средств связи с оператором.

Система должна выполнять 3 вида операций над программами пользователя или *задачами*:

1. При поступлении запроса переводить выдавшую его задачу из состояния «работа» в состояние «ожидание» и ставить задачу в очередь в соответствии с ее приоритетом.

2. Проверять очередь задач, которые находятся в состоянии «ожидание», и, выбрав из них имеющую наивысший приоритет, исключать ее из очереди и переводить в состояние «работа».

3. Передавать CPU задаче, находящейся в состоянии «работа». В общем случае такая система не требует специального оборудования. В ней могут применяться любые средства взаимодействия, определенные пользователем, которому нужно только описать задачи, оформленные как программы для PDP-11, и включить их в систему следующим образом:

1) добавить их в интервал времени между завершением одной программы и началом выполнения другой;

2) создать собственные входы в различные очереди, таблицы и программные секции (PSECT), описанные в следующем разделе.

11.2. СТРУКТУРА ОЧЕРЕДЕЙ, ТАБЛИЦ И PSECT

При описании задачи пользователь должен указать для нее один из четырех уровней программного (или пользовательского) приоритета Эти уровни являются логическими и соответствующие им значения приоритетов — ниже системных, устанавливаемых при возникновении прерываний по I/O или вызванных выполнением соответствующих команд. Три уровня программных приоритетов соответствуют реальным, их установка приводит к изменениям содержимого слова состояния процессора. Так, при возникновении прерывания, индицирующего время запуска новой задачи с приоритетом, превышающим приоритет выполняемой, выполнение текущей задачи приостанавливается и активизируется новая. Выполнение прерванной задачи будет возобновляться в периоды приостановки задачи, имеющей больший приоритет.

11.2.1. Очередь к процессору

Очередь к CPU (CPUQ) — это информационная структура, содержащая данные о том, какие задачи ждут CPU. Она представляет собой вектор из 8 байт, разделенный на 4 элемента, причем каждый из них соответствует одному из четырех разрешенных для задач приоритетных уровней, следовательно, имеет длину 2 байта; старший байт содержит идентификатор (ID) первой задачи (данного уровня приоритета) в связанном списке задач, ожидающих обслуживания (параграф 11.2.2), а младший — ID последней задачи в этом списке. Таким образом, 2 байта служат как бы начальной и конечной точками для множества связанных задач с заданным уровнем приоритета

Если на определённом уровне приоритета задач не имеется, то соответствующий элемент CPUQ будет равен 0. Это означает, что ни одной задаче, использующей данную систему, не разрешается иметь ID, равный 0, так как 0 в любом элементе CPUQ означает, что «на этом уровне приоритета нет задач, ожидающих обработки». И наоборот, если на определенном уровне приоритета ожидает только одна задача, то оба байта CPUQ для данного уровня будут содержать ID ждущей задачи, так как в этом частном случае она является одновременно первой и последней в очереди. Из сказанного следует, что каждая задача должна иметь свой собственный единственный ID. Он назначается пользователем, когда определяется PSECT для задачи (параграф 11.2.4).

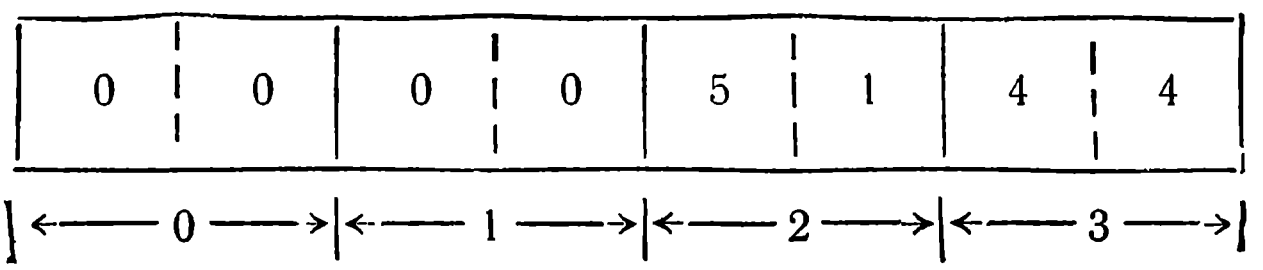


Рис. 11.1. Структура CPUQ

Структура CPUQ показана на рис. 11.1. На этом примере отражено следующее:

1) нет задач, ожидающих обслуживания, с приоритетом 0 или 1;

2) имеется список задач, ожидающих обслуживания, на уровне приоритета 2. Задача с ID = 5 — это первая задача в списке, а задача с ID = 1 — последняя;

3) на уровне 3 имеется только одна задача, ожидающая обслуживания. Ее ID = 4.

Обслуживание задач идет от высшего уровня к низшему. Таким образом, задача с ID = 4 будет обслужена первой.

11.2.2. Таблица очереди к CPU

Таблица очереди к CPU, CPUQTB есть вектор, содержащий столько элементов (слов), сколько задач имеется в системе. (В действительности число элементов равняется числу задач +1, поскольку 0-й элемент не используется по причинам, которые будут кратко изложены ниже.) CPUQTB — это связанный список ID задач, причем в нем объединены задачи с одинаковым уровнем приоритета. Поскольку имеются 4 уровня приоритета, возможно одновременное существование четырех связанных списков CPUQTB.

Допустим, что в системе имеется N задач (максимальное N = 255). Тогда структура CPUQTB примет вид, показанный на рис. 11.2. Как было установлено выше, каждая задача имеет свой идентификационный номер, или ID, который был ей присвоен при создании PSECT. Эти ID идут в порядке возрастания от

1 до наибольшего номера задачи, используемого системой. Таким образом, если нужно определить 3 задачи, то они будут иметь ID, равные 1, 2 и 3. (Следует помнить, что ID = 0 запрещен, так как 0 интерпретируется супервизором как «отсутствие ждущей задачи» на данном уровне приоритета.)

Очевидно, что каждый элемент CPUQTB соответствует ID задачи, ожидающей обслуживания. Например, элемент 1 соответствует задаче с ID = 1, элемент 2 — задаче с ID = 2 и т. д. (Разумеется, элемент 0 не используется, так как ни одна из действительных задач не может иметь ID = 0.) Обратившись к содержимому каждого элемента, мы увидим, что слова делятся на 2 байта, причем каждый байт хранит ссылку «вперед» или «назад» в связанном списке. Здесь указатели — это не адреса, а ID задач.

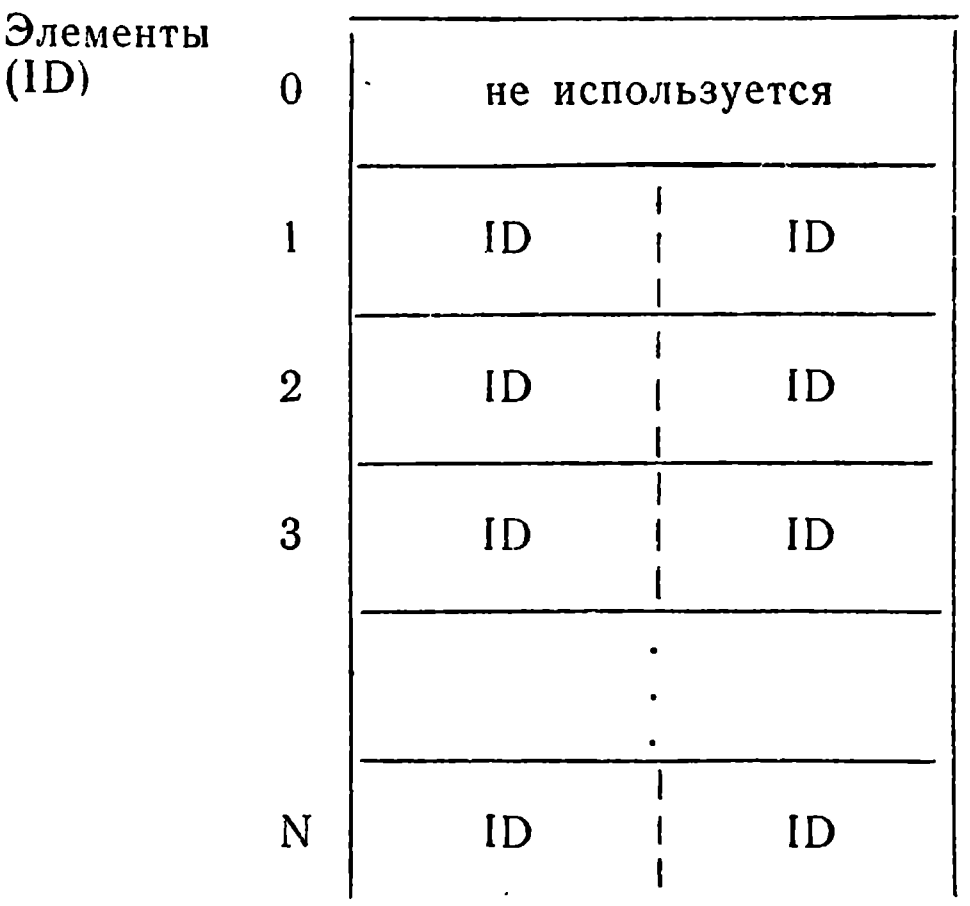


Рис. 11.2. Структура таблицы очереди к CPU

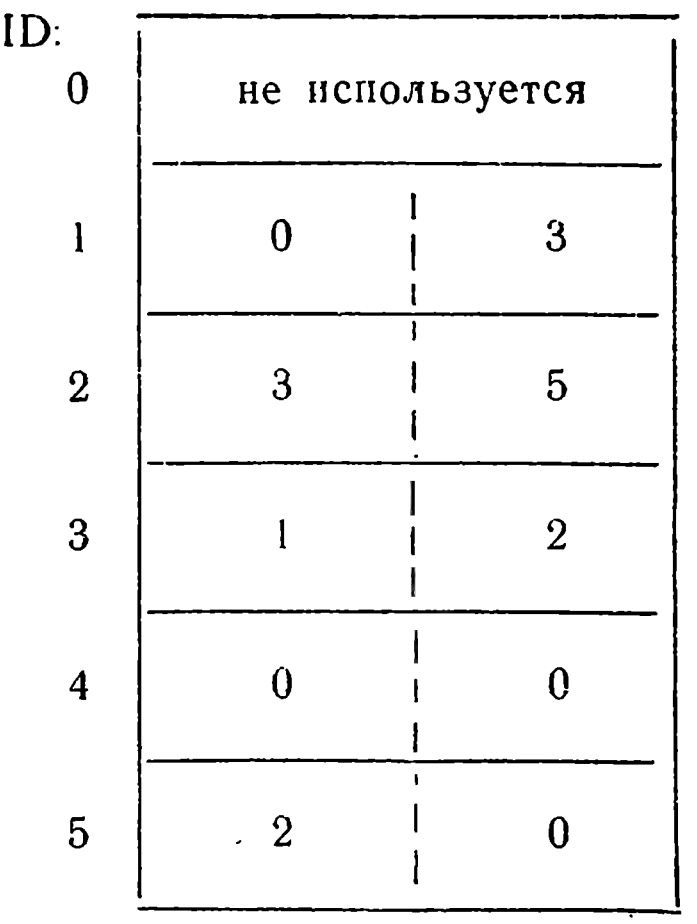


Рис. 11.3. Пример элементов CPUQTB

Для пояснения последнего положения приведем следующий пример. Допустим, что имеются 4 задачи с идентификаторами, равными 5, 2, 3 и 1, все они имеют приоритет 2 и ожидают обслуживания в очереди. Далее, имеется задача с ID = 4 на уровне 3, также ожидающая обслуживания. На рис. 11.1 представлена CPUQ для этих 5 задач, а на рис. 11.3 — CPUQTB. Как показано на рис. 11.1, первая задача на уровне 2 — это задача с ID = 5, последняя задача в списке — это задача с ID = 1.

Как показано на рис. 11.3, элемент для задачи с ID = 5, делится на 2 байта. Старший байт содержит ID = 2, что означает: «задача с ID = 2 следует за задачей с ID = 5 в этом связанном списке». Младший байт содержит 0, означающий «нет задач, предшествующих задаче с ID = 5» или «задача с ID = 5 стоит во главе списка». Далее, элемент 2 CPUQTB соответствует задаче с ID = 2, на этом входе располагается следующая информация:

старший байт равен 3, младший байт равен 5, что означает: «задача с ID = 5 предшествует, а задача с ID = 3 следует за задачей с ID = 2». И наконец, элемент 1 CPUQTB предназначен для задачи с ID = 1. Он содержит 0 и 3 соответственно в старшем и младшем байтах, что означает: «задача с ID = 3 предшествует задаче с ID = 1, а задач, следующих за ней, не имеется». Таким образом, задача с ID = 1 — это последняя задача в списке.

Обратившись к таблице для задачи с ID = 4, читатель увидит, что оба байта для этого входа будут содержать 0. Это объясняется тем, что задаче с ID = 4 не предшествует и за ней не следует никакая другая задача в очереди (т. е. эта задача — единственная, ждущая в очереди к процессору на уровне приоритета 3).

11.2.3. Таблица PSECT

PSECT («секция процесса», где *процесс* является синонимом задачи; см. параграф 11.2.4) — это вектор длиной в 11 слов; он создается пользователем для каждой задачи. Информация, находящаяся в PSECT, используется супервизором при переключениях задач. Однако прежде, чем описывать PSECT, мы рассмотрим структуру таблицы PSECT, PSECTB.

PSECTB — таблица, содержащая один элемент для каждой PSECT. Поскольку требуется одна секция PSECT для каждой задачи, таблица PSECTB будет иметь столько элементов, сколько имеется задач. Но, поскольку элемент 0 в таблице PSECTB не используется, PSECTB аналогично CPUQTB будет содержать число элементов, равное количеству задач + 1.

ID:	
0	не используется
1	Psect 1
2	Psect 2
3	Psect 3
4	Psect 4
5	Psect 5

Рис. 11.4.
Пример элементов PSECTB

Каждый элемент PSECTB соответствует ID задачи, адрес PSECT которой будет в нем храниться. Пример структуры такой таблицы показан на рис. 11.2.4. Элемент 0 не содержит информации, так как не может существовать задачи с ID=0. Остающиеся 5 элементов соответствуют задачам с ID от 1 до 5, адреса PSECT которых задаются в символическом виде: от PSECT 1 до PSECT 5 соответственно.

На этом этапе читатель может определить размер таблицы PSECT и использовать ID вместо истинных адресов для любых задач в системе. Благодаря этому остается больше места для элементов таблиц (каждый из которых занимает 1 байт) и тем самым сокращается объем требуемой для системы памяти. В то

же время упрощается обработка этих таблиц, так как используются только линейные списки в векторной форме, и, как следствие, доступ к элементу требует только двух шагов: (1) изменить величину индекса для выравнивания байта на границу слова и (2) добавить базовый адрес таблицы для того, чтобы вычислить действительный вход в PSECTB, как это показано ниже.

```
ASL     R0              ;CONVERT INDEX TO WORD OFFSET
ADD     #PSECTB,R0      ;ADD IN TABLE BASE ADDRESS
```

11.2.4. Вектор PSECT

Во время ассемблирования пользователь резервирует 11 слов в векторе PSECT. Этот вектор идентифицируется адресом первого слова, и его адрес указывается в соответствующем элементе таблицы PSECTB. PSECT содержит информацию, касающуюся только данной задачи, включая ее стек, в котором сохраняется содержимое регистров, пока задача ожидает своей очереди на выполнение.

Структура PSECT показана на рис. 11.5. Младший байт первого слова PSECT содержит номер ID данной задачи, старший

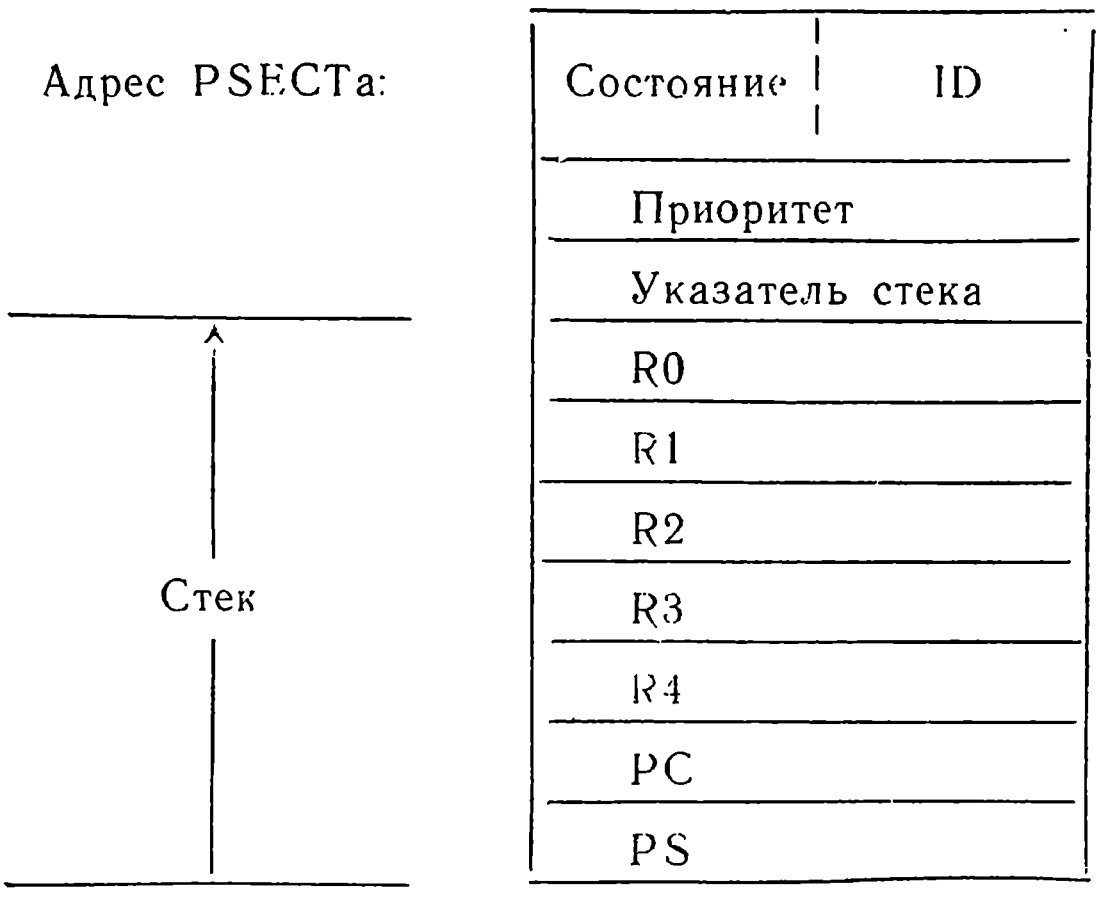


Рис. 11.5. Структура PSECT

байт отражает состояние задачи. Задача может находиться в одном из двух состояний: «работа», когда она владеет CPU, и «ожидание», если задача ждет своей очереди. Состояния можно идентифицировать использованием индивидуальных бит для обозначения различных состояний (до 8 бит) или кодированием состояния в виде цифровой величины (до 256). Как уже отмечалось, в данной простой системе имеются два состояния: «работа» = 1 и «ожидание» = 2.

Второе слово PSECT содержит значение приоритета. Ранее оговаривалось, что задача может выполняться на одном из четырех уровней приоритета (0, 1, 2, 3). Биты приоритета располагают-

ся в соответствии со структурой слова состояния процессора (PSW), т. е. биты 5, 6, 7 образуют приоритет. Поэтому если задачу нужно выполнить на уровне приоритета 2, то второе слово ее PSECT должно содержать восьмеричное число 100.

В третьем слове PSECT сохраняется указатель стека задачи. Остальные поля PSECT (8 слов) используются как стек, где будет запоминаться содержимое регистров PC и PS, при переключении задачи из состояния «работа» в состояние «ожидание». Таким образом, PSECT можно сформировать следующим образом:

```
PSECT1: .BYTE     1,1       ;STATE AND ID BYTES
        .WORD     100       ;PRIORITY WORD, LEVEL 2
        .WORD     0         ;STACK POINTER SAVED HERE
        .BLKW     6         ;STACK AREA FOR R0-R5
        .WORD     TASK1     ;PC FOR TASK 1
        .WORD     0         ;PS FOR TASK 1
```

Если стек находится в пределах области PSECT, то при работе с ним возникают некоторые ограничения. Во-первых, если пользователь намеревается использовать свой стек во время состояния «работа», то в нем нельзя разместить больше девяти слов. Во-вторых, в тот момент, когда задача освобождает процессор, указатель стека, шестой регистр, должен быть загружен своим первоначальным значением #PSECT + 26 (т. е. значение вершины стека). При этом в то время, когда задача находится в состоянии ожидания, в стеке не должно оставаться ни одного запомненного слова. Эти ограничения можно снять, что и оставляем в качестве упражнения для читателя (см. упражнение 2 в конце главы).

11.3. СУПЕРВИЗОР MMS

Супервизор является основным элементом системы MMS. Он может быть вызван любой выполняющейся задачей с помощью команды супервизора QUEUE. *QUEUE* — это инструкция, вызывающая прерывание (CUEUE = 104000), по которой будет считываться вектор размером в два слова, начиная с ячейки 30 (8) *. Первое его слово содержит адрес головной программы из программ супервизора, которые будут выполняться в заданной очередности. Второе слово вектора содержит приоритет, в соответствии с которым выполняются программы супервизора.

Супервизор включает следующие пять программ (в порядке их выполнения):

1. SCHEDL — программа-планировщик;
2. SAVE — программа сохранения регистров;
3. INSLAS — программа постановки в очередь;
4. DISPCH — программа-диспетчер;
5. UNCHN — программа отсоединения от очереди.

Эти программы описаны ниже.

---

* Эта инструкция в реальных ЭВМ и операционных системах соответствует инструкции EMT. — *Примеч. ред.*

11.3.1. Программа-планировщик

В результате команды супервизора QUEUE (т. е. SVC) управление переходит к программе-планировщику, SCHEDL. Следовательно, первое слово вектора прерывания по обращению к супервизору должно содержать адрес SCHEDL. В результате выполнения команды QUEUE SVC состояние процесса и счетчик команд автоматически передаются в частный стек задачи, выдавшей SVC. После этого первым действием SCHEDL является сохранение (путем вызова программы сохранения SAVE) содержимого всех регистров, т. е. запоминание их в стеке пользователя.

Затем, используя PSECT задачи, выдавшей SVC, SCHEDL определяет ее приоритет и находит нужный элемент CPUQ. Затем (с помощью программы INSLAS) задача помещается в конец очереди (CPUQTB) и переключается из состояния «работа» в состояние «ожидание». В завершение выбирается новая задача, которой передается контроль над CPU (за счет работы программы-диспетчера DISPCH).

11.3.2. Программа сохранения регистров

Программа SAVE вызывается из программы SCHEDL. Поскольку R5 уже помещен в стек командой JSR, требуется сохранить только оставшиеся регистры от R4 до R1. На этом этапе сохраняется указатель стека пользователя, а адрес активного PSECT помещается в регистр 5. Наконец, система переключается в супервизорное состояние и при помощи подпрограммы SUPSTK формирует новый стек. При возврате из этой подпрограммы не должна выполняться команда JSR, поскольку новый стек не содержит адреса возврата. Вместо этого применяется команда перехода, в которой используется адрес возврата, содержащийся в регистре 5. Команда перехода имеет еще одно дополнительное преимущество — она не производит выталкивания из стека, так как это противоречило бы одной из целей подпрограммы SAVE.

11.3.3. Программа постановки в очередь

SCHEDL вызывает программу INSLAS после того, как вычислен адрес элемента CPUQ, соответствующего вызывающей задаче. INSLAS должна «поместить» задачу, выдавшую SVC, в предназначенный ей список ожидания. Для этого INSLAS выполняет следующие действия.

Во-первых, INSLAS получает ID задачи из ее PSECT и с помощью этого идентификатора вычисляет адрес соответствующего элемента CPUQTB. Поскольку задача станет самой последней в очереди, старший байт элемента CPUQTB будет равен 0 (т. е. за данной задачей в списке ожидающих не следует никакая другая). По известному значению приоритета задачи INSLAS проверяет содержимое соответствующего элемента очереди CPUQ с целью определения наличия в ней других задач. Если очередь

CPUQ:

Неизвестно	Неизвестно	0 : 0	Неизвестно
0	1	2	3

CPUQTB:

0	Не используется
1	Неизвестно
2	Неизвестно
3	Неизвестно
4	Неизвестно
5	Неизвестно
6	Неизвестно

(*а*) До

CPUQ:

Неизвестно	Неизвестно	4 : 4	Неизвестно
0	1	2	3

CPUQTB:

0	Не используется
1	Неизвестно
2	Неизвестно
3	Неизвестно
4	0 : 0
5	Неизвестно
6	Неизвестно

(*б*) После

CPUQ:

Неизвестен	Неизвестно	3 : 5	Неизвестен
0	1	2	3

CPUQTB:

0	Не используется
1	5 : 6
2	Неизвестно
3	6 : 0
4	Неизвестно
5	0 : 1
6	1 : 3

(*в*) До

CPUQ:

Неизвестно	Неизвестно	3 : 4	Неизвестно
0	1	2	3

CPUQTB:

0	Не используется
1	5 : 6
2	Неизвестно
3	6 : 0
4	0 : 5
5	4 : 1
6	1 : 3

(*г*) После

Рис. 11.6. Структура CPUQ и CPUQTB до и после выполнения программы INSLAS

свободна (т. е. элемент содержит нули), то данная задача будет единственной в очереди, и INSLAS обнуляет соответствующий элемент CPUQTB. После этого запоминается ID задачи в обоих байтах элемента в CPUQ.

Если очередь занята (т. е. вход в CPUQ не равен 0), это означает, что на данном уровне приоритета уже имеется одна или более задач в состоянии ожидания. Тогда INSLAS должна поместить информацию о новой задаче в различные очереди, для чего она осуществляет следующие шаги:

1) берет содержимое младшего байта элемента CPUQ. Этот байт хранит ID последней задачи (данного уровня приоритета), ожидающей в очереди;

2) определяет адрес элемента CPUQTB, относящегося к этому ID. Старший байт элемента равен 0, так как в данный момент это последняя задача, ожидающая в очереди;

3) записывает ID задачи, помещаемой в очередь, в старший байт такого элемента CPUQTB;

4) переносит старое содержимое младшего байта элемента CPUQ в младший байт элемента CPUQTB, соответствующего новой задаче;

5) помещает ID новой задачи в младший байт элемента CPUQ.

На рис. 11.6 изображены действия по постановке в очередь задачи, ID которой равен 4, а уровень приоритета — 2. В первом случае (рис. 11.6 (а) и (б)) показано, что в очереди нет каких-либо других задач с приоритетом 2. Во втором случае (рис. 11.6 (в) и (г)) предполагается, что на уровне приоритета 2 в очереди ждут 4 задачи, имеющие ID 3, 6, 1 и 5.

11.3.4. Программа-диспетчер

После того как задача поставлена в очередь, она переводится в состояние ожидания, а в качестве кандидата на выполнение выбирается другая. Программа, выбирающая эту новую задачу, представляет собой программу-диспетчер DISPCH, вызываемую из SCHEDL. Диспетчер выполняет свои функции, осуществляя поиск в CPUQ (от высшего уровня приоритета до низшего) любых задач, которые могут стать активными. Если такая программа найдена, DISPCH вызывает программу отсоединения от очереди, UNCHN, для того, чтобы отсоединить выбранную задачу из очереди.

После такого действия указатель ACTIVE обновляется с тем, чтобы указывать PSECT вновь избранной задачи. При этом PSECT индицирует, что задача находится в состоянии «работа», а ее указатель стека помещается в регистр SP. Наконец, содержимое регистров задачи восстанавливается, и DISPCH выдает команду RTI (возврат из прерывания), по которой восстанавливается содержимое PC и PS задачи. На этом этапе задача получает управление процессором.

11.3.5. Программа отсоединения от очереди

Программа отсоединения от очереди, UNCHN, вызываемая из DISPCH, извлекает первую ожидающую задачу из очереди и изменяет содержимое элементов CPUQ и CPUQTB. Действия UNCHN обратны действиям программы INSLAS. Следует помнить, что очередь обслуживается в соответствии с дисциплиной «первым вошел, первым вышел».

Кроме того, что программа UNCHN корректирует содержимое элементов очередей, она является источником адреса PSECT отсоединенной задачи. По окончании работы UNCHN возвращает управление в DISPCH.

11.4. НАПИСАНИЕ ПРОГРАММ ДЛЯ ВЫПОЛНЕНИЯ ПОД УПРАВЛЕНИЕМ MMS

Хотя на программы, выполняемые под управлением MMS, не накладывается каких-либо ограничений, для ее нормального

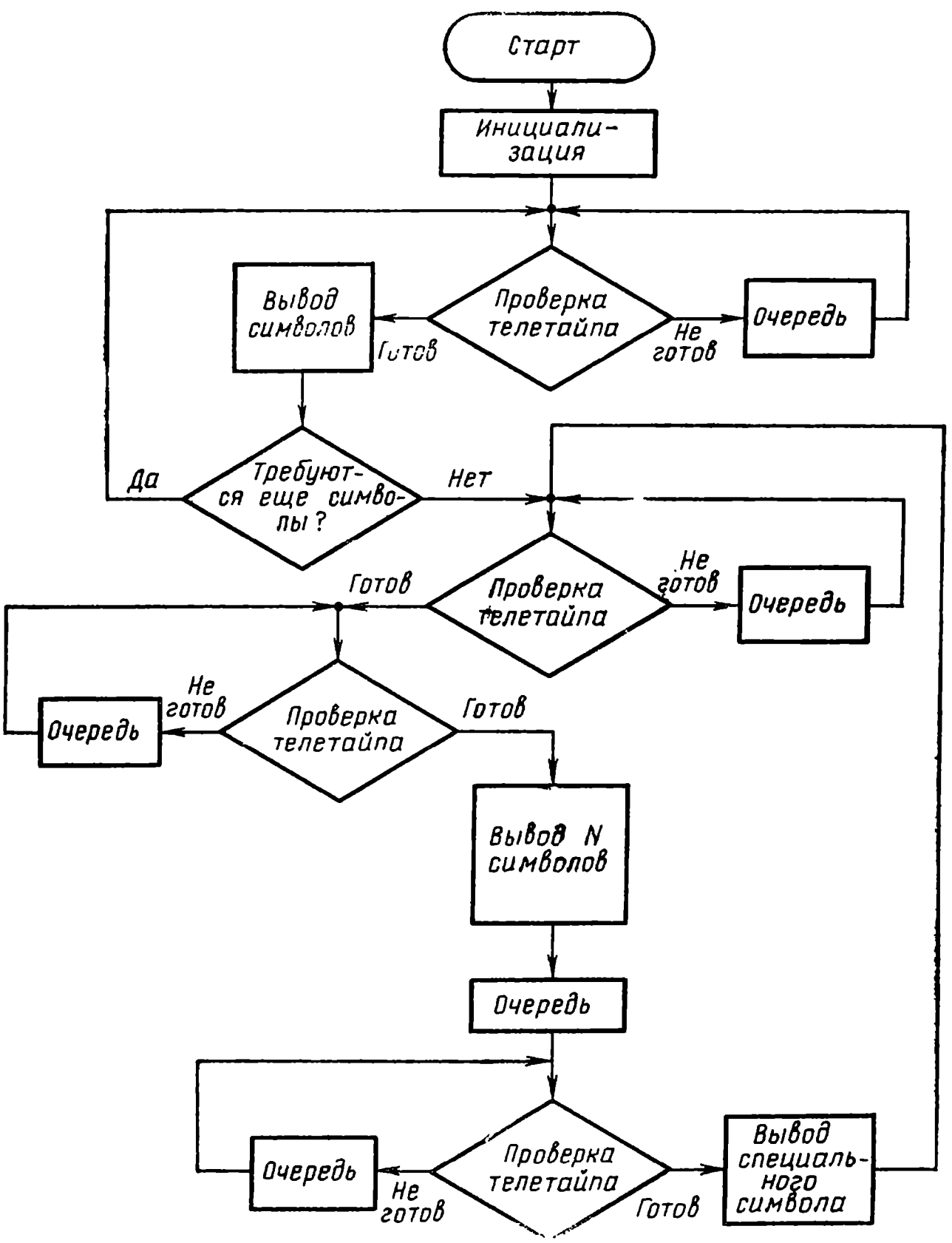


Рис. 11.7. Логическая структура примера выполнения задач в системе MMS

функционирования необходима правильная инициализация таких программ. В данном случае это означает, что пользователь должен корректно создать элементы очереди к CPU, таблицы очереди CPUQTB и таблицы PSECTB, а также сформировать описатели секций PSECT для каждой задачи, которая будет выполняться в системе (см. параграф 11.2.4). После этого пользователь может запустить MMS, установив слово состояния процессора и указатель пользовательского стека. Кроме того, необходимо занести адрес некоторого описателя PSECT в указатель активной задачи, после чего на нее будет передано управление.

В конце данной главы приводится листинг работающей системы MMS. Здесь же в качестве примеров даются две задачи: «TASK 1» и «TASK 2», демонстрирующие применение MMS. Они почти идентичны и имеют логическую структуру, показанную на рис. 11.7. Задачи осуществляют свой собственный I/O, без использования прерываний. Каждая из них проверяет состояние устройства, и если оно не готово к выполнению операции, задача сама становится в очередь, освобождая CPU.

11.5. РАСШИРЕННАЯ СИСТЕМА

Как было показано, система MMS интерпретирует только один тип команды — команду супервизора QUEUE, и по ней помещает задачу, выдавшую SVC, в конец списка ожидающих задач, а той из них, которая находится в начале этого списка, предоставляет CPU. Приложив незначительные усилия, можно реализовать другие типы команд супервизора. При этом очевидным кандидатом будет команда «WAIT», по которой соответствующая задача, будет поставлена в конец списка ожидающих. Задачи этого списка не будут помещаться в очередь к процессору (т. е. им не планируется CPU) до тех пор, пока не произойдет некоторое внешнее событие, например получение сигнала прерывания.

Применение команды супервизора WAIT SVC позволит ускорить операции с устройствами I/O, так как задача, ожидающая ввода или вывода, не получит управления процессором, пока не будет готово или не освободится требуемое устройство. В противоположность этому в рассмотренной системе любая задача, находящаяся во главе очереди к процессору, получит CPU. Далее она должна проверять состояние устройства I/O и при необходимости снова встать в очередь, если оно не готово. В тех случаях, когда имеются задачи, ожидающие I/O и CPU, бессмысленно отбирать процессор у какой-либо из задач до освобождения устройства I/O.

Обычно малые мультипрограммные операционные системы включают следующие три компонента: систему программирования I/O (IOPS), интерпретатор языка команд и процессор ошибок. Система IOPS уже рассматривалась в гл. 7. Те же принципы работы можно применить и для MMS. Небольшой и эффективный

интерпретатор команд позволит пользователю скорее загружать и прогонять задачи, запуская их с терминала, вместо того, чтобы встраивать их в саму систему. В частности, полезными могут быть команды, дающие пользователю возможность проверять и изменять содержание памяти. Наконец, программы обнаружения ошибок необходимы в том случае, если система должна обладать способностью продолжать работу в случае ошибки, когда требуется определенное действие со стороны системы (пользователя), или если какое-то непредвиденное условие требует немедленного к себе внимания.

11.6. ЛИСТИНГ MMS

```
        .TITLE MMS
        M M S - - A MODEST MULTIPROGRAMMING SYSTEM
        WRITTEN BY ED MACHADO AND MODIFIED BY R. ECKHOUSE AND R. MORRIS

        .CSECT

        D A T A    D E F I N I T I O N S

        QUEUE=104000    ;SUPERVISOR CALL TRAP
        RUNBIT=1        ;RUN STATE BIT
        PENBIT=2        ;PENDING STATE BIT
        TKSTAT=1        ;TASK STATE OFFSET
        PRIO=2          ;PRIORITY OFFSET
        TKSP=4          ;STACK POINTER OFFSET
        STATMK=17       ;STATE MASK
        IOF=340         ;TO TURN OFF INTERRUPT
        HBM=177400      ;HIGH BIT MASK
        PSW=177776      ;PROCESSOR STATUS WORD
        USERST=TASK1    ;STARTING ADDRESS OF USER TASK

        D E V I C E    D E F I N I T I O N S

        TKS1=177560     ;KEYBOARD 1 STATUS REG
        TKB1=TKS1+2     ;KEYBOARD 1 BUFFER
        TPS1=TKS1+4     ;TELEPRINTER 1 STATUS REG
        TPB1=TKS1+6     ;TELEPRINTER 1 BUFFER
        TKS2=175610     ;KEYBOARD 2 STATUS REG
        TKB2=TKS2+2     ;KEYBOARD 2 BUFFER
        TPS2=TKS2+4     ;TELEPRINTER 2 STATUS REG
        TPB2=TKS2+6     ;TELEPRINTER 2 BUFFER

        C P U    Q U E U E    D E F I N I T I O N

        THERE ARE FOUR ENTRIES (EIGHT BYTES) IN
        WHICH ENTRY 2 (PRIORITY 2) HAS THE ID
        FOR THE ONLY TASK (TASK2) WAITING ON THE QUEUE

        NOTE THAT .BYTES ARE OFFSET TO VISUALLY SUGGEST HIGH  LOW BYTES

CPUQ:
                .BYTE   0       ;ID OF LAST TASK AT LEVEL 0
        .BYTE   0               ;ID OF 1'ST TASK AT LEVEL 0
                .BYTE   0       ;ID OF LAST TASK AT LEVEL 1
        .BYTE   0               ;ID OF 1'ST TASK AT LEVEL 1
                .BYTE   2       ;ID OF LAST TASK AT LEVEL 2
        .BYTE   2               ;ID OF 1'ST TASK AT LEVEL 2
                .BYTE   0       ;ID OF LAST TASK AT LEVEL 3
        .BYTE   0               ;ID OF 1'ST TASK AT LEVEL 3

        C P U    Q U E U E    T A B L E    D E F I N I T I O N
```

```
;       ENTRY 2 IS THE ONLY ENTRY STORING RELEVANT
;       INFORMATION.  IT CONTAINS ZEROS BECAUSE THE
;       TASK WITH ID=2 (TASK2) IS NOT PRECEDED OR
;       FOLLOWED BY ANY OTHER TASK ON THE QUEUE, E.G.
;       TASK WITH ID=2 IS THE ONLY TASK WAITING FOR THE CPU
;
;       NOTE THAT .BYTES ARE OFFSET TO VISUALLY SUGGEST HIGH, LOW BYTES
;
CPUQTB: .BYTE   0,0
;
                .BYTE   0               ;ID OF TASK BEFORE TASK1
        .BYTE   0                       ;ID OF TASK AFTER TASK 1
;
                .BYTE   0               ;ID OF TASK BEFORE TASK2
        .BYTE   0                       ;ID OF TASK AFTER TASK2
;
                .BYTE   0               ;ID OF TASK BEFORE TASK3
        .BYTE   0                       ;ID OF TASK AFTER TASK3
;
                .BYTE   0               ;ID OF TASK BEFORE TASK4
        .BYTE   0                       ;ID OF TASK AFTER TASK4


;
;       P S E C T    T A B L E    D E F I N I T I O N
;
;       THE PSECTB IS INITIALIZED WITH THE ADDRESS OF
;       THE PSECT'S (PSECT1 AND PSECT2) DEFINED FOR
;       THE TASKS WITHIN THE SYSTEM (TASK1 AND TASK2)
;
PSECTB: .WORD   0
        .WORD   PSECT1                  ;ADDRESS OF PSECT FOR TASK1
        .WORD   PSECT2                  ;ADDRESS OF PSECT FOR TASK2
        .WORD   0                       ;ADDRESS OF PSECT FOR TASK3
        .WORD   0                       ;ADDRESS OF PSECT FOR TASK4

;       P O I N T E R S    A N D    T E M P O R A R I E S
;
;       ACTIVE HAS (IS INITIALIZED WITH) THE ADDRESS OF
;       THE PSECT FOR THE FIRST TASK TO GAIN CONTROL OF THE CPU
;
ACTIVE: .WORD   PSECT1
;
;       SUPSTK IS THE STACK USED BY THE SUPERVISOR WHILE IN CONTROL
;
        .BLKW   20.
SUPSTK: .WORD   0               ;STACK'S TOP CELL
;
;       P S E C T    D E F I N I T I O N S
;
;       NOTE THAT .BYTES ARE OFFSET TO VISUALLY SUGGEST HIGH, LOW BYTES
;
PSECT1:         .BYTE 1         ;ID FOR TASK1
        .BYTE   1               ;STATE FOR TASK1
        .WORD   100             ;PRIORITY OF 2
        .WORD   0               ;STACK POINTER WILL BE SAVED HERE
        .BLKW   6               ;STACK AREA FOR R0-R5
        .WORD   TASK1           ;PC FOR TASK1
        .WORD   0               ;PSW FOR TASK1 WILL BE SAVED HERE
;
PSECT2:         .BYTE 2         ;ID FOR TASK2
        .BYTE   2               ;STATE FOR TASK2
        .WORD   100             ;PRIORITY OF 2
        .WORD   0               ;STACK POINTER WILL BE SAVED HERE
        .BLKW   6               ;STACK AREA FOR R0-R5
        .WORD   TASK2           ;PC FOR TASK2
        .WORD   0               ;PSW FOR TASK2 WILL BE SAVED HERE
;
;       INITIALIZE THE SYSTEM
;
START:  CLR     @#177546                ;TURN OFF LINE CLOCK
        BIC     #100,@#TKS1             ;DISABLE INTERRUPTS FOR DEVICE 1
        BIC     #100,@#TKS2             ;DISABLE INTERRUPTS FOR DEVICE 2
```

```
        CLR     R0
ST1:    INC     R0                  ;INITIALIZE
        INC     R0                  ;ALL
        MOV     R0,-2(R0)           ;INTERRUPT VECTORS
        CLR     (R0)+               ;TO HAVE THE PC
        CMP     #400,R0             ;POINT TO THE NEXT LOCATION
        BNE     ST1                 ;WHICH IS A HALT INSTRUCTION


;
;       SET UP THE TRAP VECTOR
;
        MOV     #SCHEDL,@#30        ;FOR SVC TRAP
        MOV     #IOF,@#32           ;WITH PRIORITY 7
;
;       SET UP PSW AND STACK, THEN BEGIN
;
        CLR     PSW                 ;SET UP PROCESSOR STATUS
        MOV     #PSECT1+26,SP       ;SET UP STACK POINTER
        MOV     #PSECT1,ACTIVE      ;DEFINE THE ACTIVE TASK
        JMP     USERST              ;USER'S STARTING ADDRESS

;
;       T H E    S C H E D U L E    R O U T I N E
;
;                       "S C H E D L"
;
;       THIS IS THE SUPERVISOR ROUTINE CALLED BY A
;       TRAP THROUGH LOCATION 30 (OCTAL), IN RESPONSE
;       TO AN SVC OR QUEUE CALL.
;
SCHEDL: JSR     R5,SAVE             ;SAVE ALL REGISTERS, INCL. TSK SP
        MOV     PRIO(R5),R0         ;GET PRIORITY OF TASK TO BE QUEUED
                                    ;R5 CONTAINS ADDRESS OF PSECT FOR THAT TASK
        ASR     R0                  ;SHIFT
        ASR     R0                  ; RIGHT TO
        ASR     R0                  ;  GET CORRECT
        ASR     R0                  ;   ENTRY IN CPUQ (I.E., 2*ACTUAL PRIORITY)
        ADD     #CPUQ,R0            ;ADD IN ADDRESS OF BASE OF QUEUE TO GET
                                    ;POINTER TO CPUQ ENTRY FOR PRIORITY "N"
        JSR     PC,INSLAS           ;INSERT TASK IN QUEUE
        BICB    #STATMK,TKSTAT(R5) ;CLEAR TASK STATE OF NEWLY QUEUED ITEM
        BISB    #PENBIT,TKSTAT(R5) ;SET IT TO PENDING STATE
        JMP     DISPCH              ;HEAD OF WAITING LIST
;
;       T H E    S A V E    R O U T I N E
;
;                  "S A V E"
;
;       SUPERVISOR SUBROUTINE CALLED FROM SCHEDL WHICH
;       PUSHES THE CONTENTS OF THE USER'S REGISTERS INTO
;       THE PSECT STACK AND SWITCHES THE SYSTEM TO
;       SUPERVISOR STATE (THE SYSTEM IS IN THE SUPERVISOR
;       STATE WHEN IT IS USING THE SUPERVISOR STACK).
;
SAVE:   MOV     R4,-(SP)            ;PUSH
        MOV     R3,-(SP)            ; REMAINING
        MOV     R2,-(SP)            ;  REGISTERS
        MOV     R1,-(SP)            ;   ONTO
        MOV     R0,-(SP)            ;    STACK
        MOV     R5,RETADR           ;SET UP RETURN ADDR
        MOV     ACTIVE,R5           ;ADDRESS OF ACTIVE PSECT
        MOV     SP,TKSP(R5)         ;SAVE STACK PTR
        MOV     #SUPSTK,SP          ;SUPERVISOR STATE/STACK
        JMP     @#0                 ;RETURN
        RETADR=.-2                  ;2ND WORD OF JMP INSTR


;
;       T H E    I N S E R T    R O U T I N E
;
;                  "I N S L A S"
;
;       SUPERVISOR SUBROUTINE CALLED FROM SCHEDL TO
;       INSERT THE TASK THAT ISSUED AN SVC AT
;       THE END OF A WAITING LIST ("CPUQTB")
```

```
;
INSLAS: MOVB    @R5,R2          ;ID OF NEW LAST TASK AT LEVEL "N", NLT(N)
        BIC     #HBM,R2         ;CLEAR THE HIGH BYTE OF REGISTER
        ASL     R2              ;MULTIPLY BY TWO
        ADD     #CPUQTB,R2      ;ADDRESS OF BACKWARD PNTR BYTE FOR NLT(N)
        CLRB    1(R2)           ;CLEAR FORWARD POINTER OF CPUQTB FOR NLT(N)
        MOVB    @R0,R1          ;GET ID FOR PREVIOUS LAST TASK, PLT(N)
        BEQ     EMPTYQ          ;IS IT ZERO?
        BIC     #HBM,R1         ;NO, CLEAR HIGH BYTE OF ID WORD
        ASL     R1              ;MULTIPLY BY TWO
        ADD     #CPUQTB,R1      ;ADDRESS OF BACKWARD POINTER FOR PLT(N)
        MOVB    @R5,1(R1)       ;ID OF NLT(N) INTO FORWARD POINTER OF PLT(N)
        MOVB    @R0,@R2         ;OLD LAST TASK IS PRECEDING TASK
        MOVB    @R5,@R0         ;NEW TASK IS LAST TASK
        RTS     PC              ;RETURN
EMPTYQ: MOVB    @R5,@R0         ;TASK ID INTO LOW BYTE
        MOVB    @R5,1(R0)       ;AND HIGH BYTE OF CPUQ ENTRY
        RTS     PC              ;RETURN
;
;       T H E   D I S P A T C H   R O U T I N E
;
;                  "D I S P C H"
;
;       SUPERVISOR ROUTINE TO UNCHAIN A TASK FROM THE
;       WAITING LIST AND ASSIGN THE CPU TO IT
;
DISPCH: MOV     #CPUQ+10,R0     ;TRY HIGHEST PRIORITY
LOOP:   TST     -(R0)           ;ANYTHING TO DO?
        BEQ     LOOP            ;NO, TRY THE NEXT LEVEL
        JSR     PC,UNCHN        ;YES, UNCHAIN
        MOV     R5,ACTIVE               ;MAKE NEW TASK ACTIVE
        BICB    #STATMK,TKSTAT(R5)      ;CLEAR TASK STATE
        BISB    #RUNBIT,TKSTAT(R5)      ;SET STATE TO RUN
        MOV     TKSP(R5),SP     ;GET TASK'S STACK POINTER
        MOV     (SP)+,R0        ;RESTORE
        MOV     (SP)+,R1        ; CONTENTS
        MOV     (SP)+,R2        ;  OF
        MOV     (SP)+,R3        ;   REGISTERS
        MOV     (SP)+,R4        ;    PREVIOUSLY
        MOV     (SP)+,R5        ;     SAVED
        RTI                     ;RETURN FROM INTERRUPT

;
;       T H E   U N C H A I N   R O U T I N E
;
;                  "U N C H N"
;
;       SUPERVISOR SUBROUTINE CALLED FROM THE DISPCH ROUTINE
;       WHOSE ACTION IS TO TAKE THE FIRST TASK OUT OF A
;       WAITING LIST AND UPDATE THE ENTRIES BOTH IN CPUQ AND
;       CPUQTB.
;
UNCHN:  MOV     @R0,R1          ;ID OF FIRST TASK ON QUEUE
        SWAB    R1              ;SWAP ID'S
        BIC     #HBM,R1         ;CLEAR HIGH BYTE
        ASL     R1              ;MULTIPLY BY TWO
        MOVB    CPUQTB+1(R1),1(R0) ;NEW FIRST TASK
        BNE     ANTKS           ;THERE IS ANOTHER TASK
        CLR     @R0             ;NO MORE TASKS-ZERO OUT ENTRY
        BR      ARND            ;SKIP NEXT INSRTS
ANTKS:  MOVB    1(R0),R0        ;NEW "FIRST TASK" ON QUEUE
        BIC     #HBM,R0         ;CLEAR HIGH BYTE
        ASL     R0              ;MULTIPLY BY TWO
        CLRB    CPUQTB(R0)      ;NO TASK PROCEEDS
ARND:   CLR     CPUQTB(R1)      ;CLEAR THIS ENTRY
        MOV     PSECTB(R1),R5   ;GET PSECT'S ADDRESS
        RTS     PC              ;RETURN

;
;       "T A S K 1"
;
;       SIMPLE LITTLE TASK TO READ AND TYPE A CHARACTER,
;       COUNT TO 100 (OCTAL), AND THEN INSERT A SPACE.
;       AFTER EACH STEP, A QUEUE IS PERFORMED
;
```

```
TASK1:  MOV     BLANK,R5        ;PUT A BLANK IN R5
        MOV     #6,R1           ;LETTER COUNT
        MOV     #RDY,R2         ;BUFR POINTER
TA1:    TSTB    TPS1            ;TIME TO PRINT?
        BMI     PR1             ;YES
        QUEUE                   ;NO, GIVE UP CPU
        BR      TA1             ;TRY AGAIN
PR1:    MOVB    (R2)+,TPB1      ;PUT OUT CHARACTER
        DEC     R1              ;DECR LETTER COUNT
        BNE     TA1             ;MORE TO DO
AGN1:   TSTB    TKS1            ;ANYTHING THERE?
        BMI     YES1            ;YES, DON'T QUEUE
        QUEUE                   ;GIVE UP CPU
        BR      AGN1            ;TRY AGAINNAT
YES1:   TSTB    TPS1            ;CAN WE PRINT?
        BMI     YES2            ;YES, DON'T QUEUE
        QUEUE                   ;GIVE UP CPU
        BR      YES1            ;TRY AGAIN
YES2:   MOVB    TKB1,TPB1       ;PUT OUT CHARACTER
        MOV     #100,R0         ;LOOP COUNT
T2:     DEC     R0              ;DECR COUNT
        BNE     T2              ;LOOP UNTIL ZERO
        QUEUE                   ;TIME TO GIVE UP CPU
T3:     TSTB    TPS1            ;CAN WE SPACE?
        BMI     YES3            ;YES, GO AHEAD
        QUEUE                   ;GIVE UP CPU
        BR      T3              ;TRY AGAIN

YES3.   MOVB    R5,TPB1         ;MOVE SPACE TO BUFFER
        QUEUE                   ;GIVE UP CPU
        BR      AGN1            ;REPEAT THE WHOLE THING
BLANK:  .ASCII  /  /            ;SOME BLANKS
RDY:    .ASCII  /READY!/        ;READY MESSAGE

;
;       "T A S K 2"
;
;       THIS IS THE SAME AS TASK 1 WHICH SHOWS THAT
;       THE TWO TASKS CAN RUN CONCURRENTLY WITHOUT
;       USING INTERRUPTS AND WITHOUT THE LOSS OF INFORMATION.
;
TASK2:  MOV     DASH,R5         ;DIFFERENT SEPARATOR
        MOV     #7,R1           ;CHARACTER COUNT
        MOV     #PRO,R2         ;BUFR POINTER
TA2:    TSTB    TPS2            ;TIME TO PRINT?
        BMI     PR2             ;YES
        QUEUE                   ;NO
        BR      TA2             ;TRY AGAIN
PR2:    MOVB    (R2)+,TPB2      ;PUT OUT CHARACTER
        DEC     R1              ;DECR CHAR COUNT
        BNE     TA2             ;KEEP GOING
AGN2:   TSTB    TKS2            ;ANYTHING THERE?
        BMI     YES4            ;YES, DON'T QUEUE
        QUEUE                   ;GIVE UP CPU
        BR      AGN2            ;TRY AGAIN
YES4:   TSTB    TPS2            ;CAN WE PRINT?
        BMI     YES5            ;YES, DON'T QUEUE
        QUEUE                   ;GIVE UP CPU
        BR      YES4            ;TRY AGAIN
YES5:   MOVB    TKB2,TPB2       ;PUT OUT CHARACTER
        MOV     #200,R1         ;LOOP COUNT
T4:     DEC     R1              ;DECR COUNT
        BNE     T4              ;LOOP UNTIL ZERO
        QUEUE                   ;GIVE UP CPU
T5:     TSTB    TPS2            ;CAN WE DASH?
        BMI     YES6            ;YES, GO AHEAD
        QUEUE                   ;GIVE UP CPU
        BR      T5                      ;TRY AGAIN
YES6:   MOVB    R5,TPB2         ;MOVE DASH TO BUFFER
        QUEUE                   ;GIVE UP CPU
        BR      AGN2    ;REPEAT
DASH:   .ASCII  /--/            ;SOME DASHES
PRO:    .ASCII  /PROCEED/       ;DIFF MESSAGE
        .END    START           ;END OF ASSEMBLY
```

УПРАЖНЕНИЯ

1. Одна из трудностей, связанных с использованием системы MMS, состоит в том, что она имеет недостаточное количество состояний для эффективного управления задачами на различных уровнях приоритета. Таким образом, если одна задача находится на уровне приоритета 3, а другая — 7, то программа уровня 3 никогда не сможет передать CPU программе уровня 7, так как первая всегда выбирается из очереди к CPU. Как можно было бы изменить MMS, чтобы избежать этого ограничения?

2. Найдите альтернативное решение, подобное созданию стека PSECT в «системной области», для восстановления стека своей задачи.

3. Если система, допустим MMS, может быть написана так, что она работает без прерываний, то объясните, какое преимущество дают прерывания над использованием «опроса» или «scout's honor» (т. е. подхода, при котором каждая программа гарантирует освобождение CPU (с помощью SVC) в течение заданного временного интервала).

4. Каким образом к системе MMS нужно добавить таймер прерывания так, чтобы задачи могли бы освобождать CPU после заданного интервала времени? Поможет или помешает пользователю такое деление времени на интервалы? Нужно ли ему знать о наличии квантования?

5. Как было отмечено в разд. 11.5, команда супервизора WAIT очень полезна. Как ее можно включить в MMS? Какие другие SVC целесообразно еще добавить?

6. Нарисуйте схему IOPS для MMS. Какие типы SVC нужно добавить в MMS? Как использователь должен вызвать IOPS?

7. Какие типы команд вы реализовали бы в MMS? Какие новые системы, требующие интерпретатора команд, помимо IOPS, можно включить в MMS (например, загрузчик, текстовый редактор)?

8. MMS не имеет возможности управлять памятью. В ней возможна реализация формирования цепочки всех неразмещенных слов памяти в цепь свободных блоков. Первое слово каждого блока содержит его размер, а связь блоков поддерживается с помощью второго слова. Когда происходит обращение к памяти, цепь просматривается до тех пор, пока не будет найден блок длины, достаточной для удовлетворения запроса. Если такой блок отсутствует, то в системе происходит сбой. При обнаружении блока он исключается из цепочки (если его размер превышает затребованный объем памяти, остаток блока вновь включается в цепочку свободных блоков). Обратите внимание на то, что для создания блока как элемента цепочки необходимы по крайней мере два слова. Если блок точно равняется запрашиваемому размеру + 1, то он берется целиком. Операции по освобождению блоков сочетаются с их распределением. Когда блок освобождается, цепочка просматривается с целью поиска смежных с ним участков. Это осуществляется в два этапа. Вначале цепь сканируется для того, чтобы найти блок, непосредственно следующий за возвращаемым (хвост блока). Если такой блок найден, он исключается из цепочки и присоединяется к возвращаемому блоку. Теперь уже новый блок становится объектом поиска других, смежных. В итоге возвращаемый блок добавляется к цепи с установленным новым общим размером. Реализуйте эту систему управления памятью в MMS.

9. Какой тип программ диагностики ошибок может использоваться в MMS? Обязательно ли эти программы должны быть резидентны в памяти?

10. Спроектируйте и реализуйте в MMS файловую систему.

ПРИЛОЖЕНИЕ 1

ПУЛЬТ ОПЕРАТОРА

Пульт оператора PDP-11 позволяет осуществлять необходимое управление системой. Программы или информация могут вводиться вручную либо изменяться при помощи переключателей и клавиш на пульте. Более того, индикационные лампочки на передней панели пульта отображают состояние машины, содержание регистра адреса шины и данные на выходе магистрали данных. Хотя между пультами различных типов PDP-11 имеются некоторые отличия, в приложении они не учитываются. На примере устройства 11/40 мы рассмотрим лишь общие принципы функционирования пульта оператора. Он представлен на рис. П.1. ЭВМ 11/34 и 11/60 имеют пульты с ключевой клавиатурой, их функции описаны в соответствующих руководствах. Функции пульта 11/03

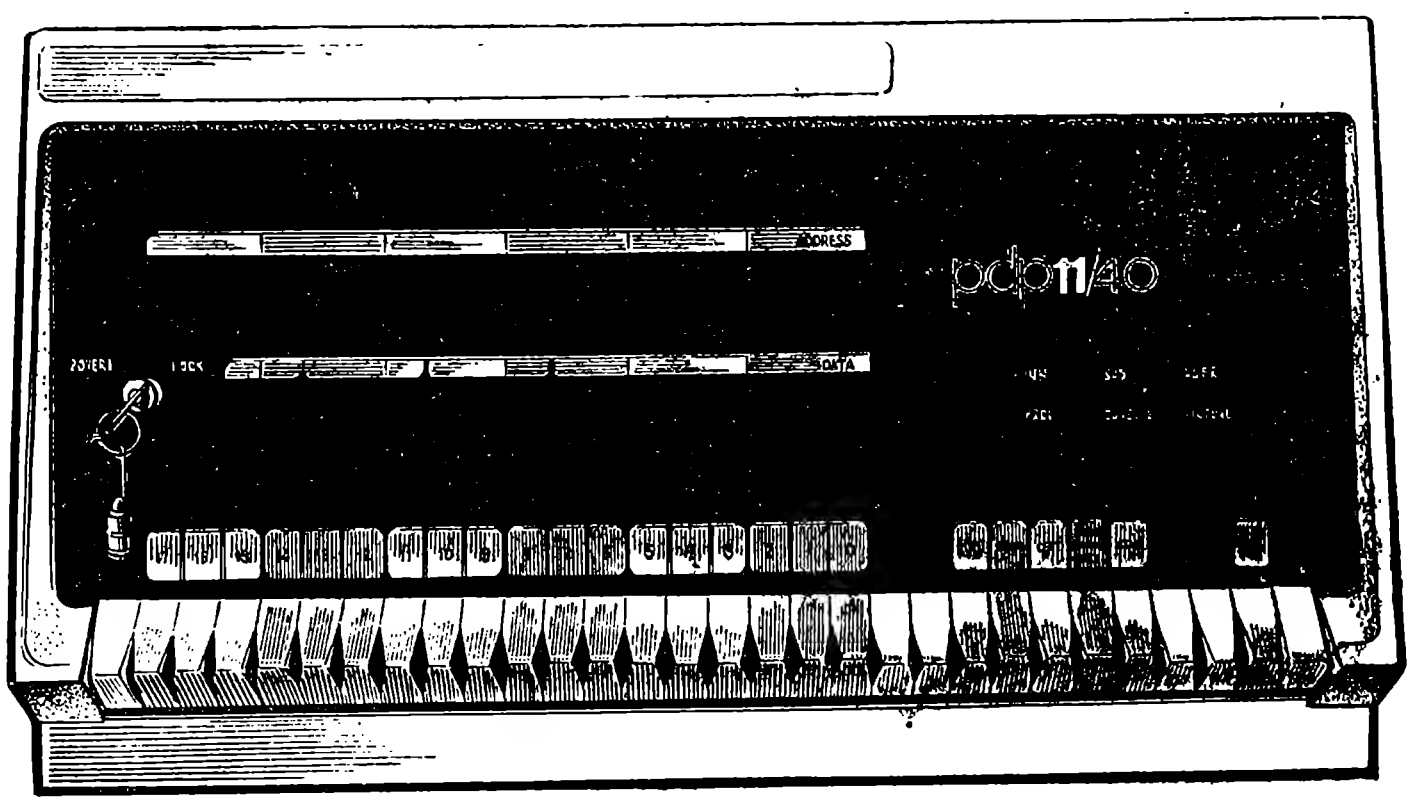


Рис. П.1. Пульт оператора PDP-11

(LSI-11) реализованы в микропрограммном мониторе, или мини-ODT. Таким образом, здесь рассматриваются пульты оператора моделей 11/05, 10, 15, 20, 35, 40, 45, 50, 55 и 70.

П.1. СОСТАВНЫЕ ЧАСТИ ПУЛЬТА

Пульт оператора имеет следующие индикаторы и переключатели:

1. Группа из 6 индикаторов, отображающих следующие условия или операции: работа, процессор, шина, пульт, пользователь и диспетчер памяти включен (виртуальный режим).
2. 18-битовый индикатор адресного регистра (регистра адреса шины).
3. 16-битовый индикатор данных.
4. 18-битовый регистр переключателей.

5. Управляющие переключатели:

a. LOAD ADRS (Load Address)
b. EXAM (Examine)
c. CONT (Continue)
d. ENABLE/HALT
e. START
f. DEP (Deposit)

П.1.1. Лампочки индикаторов

Индикаторы отражают специальные машинные функции, операции или состояния. Все они определены ниже:

1. **Работа:** лампочка горит, когда работает таймер процессора. Она погашена, когда процессор ждет асинхронного ответа от периферийного устройства или во время выполнения команды RESET. Она включена во время выполнения команд WAIT или HALT.
2. **Процессор:** лампочка загорается, когда процессор управляет шиной.
3. **Шина:** лампочка загорается, когда работает UNIBUS.
4. **Консоль:** лампочка загорается при работе в режиме пульта (работа вручную). Машина останавливается, и программа не выполняется.
5. **Пользователь:** лампочка загорается, когда CPU выполняет программу в режиме пользователя.
6. **Виртуальный режим:** лампочка загорается, когда регистр адреса показывает 16-битовый виртуальный адрес.

П.1.2. Индикаторы регистров

Пульт оператора может отобразить 18-битовый адресный регистр и 16-битовые данные. Индикатор адресного регистра непосредственно подключается к выходу 18-битового тригерного регистра, называемого «Регистр адреса шины». Этот регистр отображает адрес проверяемых или хранимых данных.

16-битовый регистр данных на передней панели пульта делится с помощью указателя на два 8-битовых байта. Этот регистр связан с выходом потока данных процессора и отражает выход сумматора процессора. После выполнения команды HALT индикатор данных будет показывать содержимое регистра R0. Кроме того, он будет показывать проверяемые данные или данные, помещаемые в память во время выполнения управляющих операций чтения или записи с пульта.

П.1.3. Регистр переключателей

PDP-11/10 и PDP-11/20 могут ссылаться на адреса до 2^{16} бит, в то время как PDP-11/40 и PDP-11/45, использующие диспетчер памяти, могут ссылаться на адреса до 2^{18} бит. Следовательно, UNIBUS сконструирована с возможностью увеличения емкости до полной 18-битовой адресации. Чтобы пульт мог обеспечивать доступ к пространству 18-битовых адресов, регистр переключателей имеет разрядность 18 бит (от 0 до 17). Два самых старших используются только как адреса. Считается, что переключатель в положении «включено» имеет значение 1, а в положении «выключено» — значение 0.

Используя 18 переключателей, можно загрузить информацию в регистр адреса шины или в какую-то ячейку памяти при помощи соответствующих управляющих переключателей, которые будут описаны ниже. Аналогично, PDP-11/70, которая может адресовать 2^{22} бита, имеет консоль с 22-битовым регистром переключателей.

П.1.4. Управляющие переключатели

Переключатели, упомянутые в разделе «Составные части пульта», выполняют следующие специальные функции управления:

1. LOAD ADRS — передает содержимое 18-битового регистра переключателей в регистр адреса шины.

2. EXAM — отражает содержимое ячеек, определенных регистром адреса шины.

3. CONT — позволяет машине продолжать работу без инициализации, в каком бы состоянии она ни находилась, при условии, что не было других команд с клавиатуры.

4. ENABLE/HALT — разрешает или запрещает выполнение программы. Для того чтобы она могла выполняться, переключатель должен находиться в положении ENABLE (вверх). Переводя переключатель в положение HALT (вниз), можно остановить систему.

5. START — запускает программу на выполнение, когда переключатель ENABLE/HALT находится в положении ENABLE. При нажатии переключателя осуществляется инициализация системы и ее запуск после того, как переключатель отпущен. Процессор начнет выполнение с адреса, который был загружен последним клавишей LOAD ADRS; при этом не возникает никакой необходимости выполнять дополнительные операции с клавиатурой. Если ENABLE/HALT находится в положении HALT, то нажатием переключателя START осуществляется сброс системы.

6. DEP — записывает содержимое 16 младших бит регистра переключателей по адресу, который отображен адресным регистром (этот переключатель работает при поднятии его вверх).

Когда в системе выполняется программа, функции LOAD ADRS, EXAM и DEPOSIT не могут предотвратить ее сбой. Если требуется остановить машину, то переключатель ENABLE/HALT переводится в позицию HALT. Машина остановится после выполнения текущей команды.

П.2. РАБОТА УПРАВЛЯЮЩИХ ПЕРЕКЛЮЧАТЕЛЕЙ

Остановив машину PDP-11, можно проверить и откорректировать ячейки шины. Для проверки специфической ячейки оператор устанавливает переключатели регистра переключателей в соответствии с ее адресом. Затем он нажимает клавишу LOAD ADRS, что приводит к записи содержимого регистра переключателей в регистр адреса шины. Адрес ячейки, которую нужно проверить, отображается при этом на индикаторах регистра адреса шины. После этого оператор нажимает клавишу EXAM. Данные, находящиеся в этой ячейке, отобразятся на индикаторе регистра данных.

При повторном нажатии клавиши EXAM содержимое регистра адреса шины увеличивается на 2, образуя адрес следующего слова, и проверяется новая ячейка. В PDP-11 регистр адреса шины всегда будет отображать текущее состояние регистра данных. При нажатии переключателя EXAM происходит увеличение адреса, после чего осуществляется проверка ячейки.

Функция проверки была сконструирована таким образом, что при нажатии клавиш LOAD ADRS и EXAM содержимое адресного регистра не меняется. В этом случае непосредственно проверяется ячейка, отраженная в индикаторе адресного регистра. Однако при втором и последующих нажатиях клавиши EXAM, содержимое регистра адреса шины увеличивается. Это продолжается до тех пор, пока не будет нажат другой управляющий переключатель.

Если оператор обнаруживает неверные данные в регистре данных, то он может ввести новые, набрав их на регистре переключателей и подняв клавишу DEP. При внесении этих данных содержимое адресного регистра не увеличится. Таким образом, нажимая клавишу EXAM, оператор может проверить только что внесенные данные. Однако, если он снова нажмет клавишу EXAM, адрес увеличится.

Если оператор попытается проверить данные, введенные в несуществующую ячейку памяти, то по «тайм ауту» шины возникнет ошибка. После этого на регистре данных высветится ячейка 4, т. е. вектор прерывания, соответствующий обращению к несуществующим адресам. Чтобы проверить это условие, оператор должен попробовать ввести какое-нибудь число (кроме 4) в ячейку, вызвавшую ошибку. Если по-прежнему высвечивается 4, то это означает, что либо в ячейку ничего не занесено, либо запись не может быть выполнена правильно.

При последовательной проверке или записи адрес будет увеличиваться на 2, выравниваясь на границу слова. Однако, если программист считывает регистры общего назначения, приращение осуществляется только на 1. Это обусловлено тем, что как только программист правильно установит регистр переключателей, он сможет использовать его 4 самых младших значащих бита для проверки общих регистров с передней панели.

Для запуска программы PDP-11 программист загружает ее начальный адрес в регистр переключателей, нажимает клавишу LOAD ADRS и, убедившись, что переключатель ENABLE/HALT находится в положении ENABLE, нажимает клавишу START. Программа начинает выполнение как только будет отпущена клавиша START. Лампочка индикатора работы загорается. Обычно, когда система работает, горит не только эта лампочка, но и другие (RUN, PROCESSOR, BUS, CONSOLE, USER, лампочки адресов, а также адресный регистр и регистр данных). Если светится только индикатор работы, а другие индикаторы не мерцают, то система, по всей видимости, выполняет команду WAIT. В этом случае на индикаторе данных отображается «1»

При режиме HALT, если оператору нужно выполнить единичную команду, он должен нажать клавишу CONT. Когда клавиша CONT нажата, пульт передает управление процессору, разрешая машине выполнить команду. Всякий раз при нажатии клавиши CONT ЭВМ будет выполнять только одну команду. Регистр адреса шины высветит последний адрес, на который ссылалась команда (но не обязательно адрес ее самой), а индикатор данных отразит содержимое этого адреса.

Для того чтобы процессор вновь начал выполнять программу, переключатель ENABLE/HALT устанавливают в положение ENABLE и нажимают клавишу CONT.

ДОПОЛНЕНИЕ К РУССКОМУ ИЗДАНИЮ: ОБЗОР МОДЕЛЕЙ ЭВМ СЕМЕЙСТВА PDP-11 И ИХ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ*

К 1982 г. история ЭВМ распространенного семейства PDP-11 насчитывала 12 лет. Этот период характеризовался постоянным совершенствованием их на основе развития полупроводниковой технологии и созданием новых средств программного обеспечения. Выпускаемые сегодня машины моделей 11/04, 11/34, 11/24, 11/44 и 11/70 сохраняют такие основные признаки, как длина слова, режимы адресации, система команд, использование шины UNIBUS для подключения периферийных устройств и ряд других, обеспечивающих программную совместимость машин этого семейства. В то же время базовая архитектура, представленная на рис. 3.1, в новых моделях подверглась существенному развитию.

За этот же период значительным изменениям подверглось и программное обеспечение ЭВМ семейства PDP-11.

Поэтому представляется целесообразным кратко рассмотреть современные технические и программные средства PDP-11. Мы уделим внимание прежде всего тем из них, которые не обсуждены в данной книге.

Д.1. НОВЫЕ ЭЛЕМЕНТЫ АРХИТЕКТУРЫ

В первую очередь с точки зрения архитектуры стоит отметить средства защиты и управления памятью, обеспечивающие доступ к адресному пространству размером до 4 Мбайт (11/70). Повышение производительности в данном случае достигается за счет совершенствования собственно CPU, а также путем введения дополнительных специализированных функциональных блоков. В разд. 3.5.2 перечислены дополнительные по отношению к базовой архитектуре элементы, существовавшие к моменту выхода книги П. Экхауза и Л. Морриса. Позже в моделях 11/24 и 11/44 появилось еще одно аппаратно реализуемое средство — *подмножество системы команд, ориентированное на обработку символьной информации и десятичных целых чисел (CIS-11)*.

Команды, входящие в состав CIS-11, могут иметь 2 формата:

регистровый, при котором операнды размещаются в регистрах общего назначения;

строчной (in-line), когда сами операнды или указатели на них размещаются в ячейках памяти, непосредственно следующих за командой. Команды этого типа отличаются прибавлением буквы «I» к мнемонике кода операции.

Возможны три типа символьных операндов:

1. *Символ.* Он занимает младшие 8 бит в регистре общего назначения, причем старшие 8 бит должны быть нулевыми.

2. *Строка символов.* Это последовательность байтов, в каждом из которых упаковано по одному символу. Старший символ строки упаковывается в байте с наименьшим адресом. Для адресации к строке символов используют два дескриптора: первый содержит адрес старшего символа, а второй — число символов в строке в пределах от 0 до 65535. В зависимости от формата команды дескрипторы размещаются либо в двух следующих за командой ячейках, либо в двух последовательных регистрах.

3. *Таблица символов.* Использование таблиц символов позволяет произвольным образом разбивать множество из 256 символов на 8 ортогональных подмножеств. Любая такая таблица состоит из 256 байт, и каждому из символов

* Обзор составлен по зарубежным источникам. — *Примеч. ред.*

в ней соответствует байт, номер которого равен коду символа. Принадлежность символа к одному из подмножеств указывается занесением единицы в соответствующий бит этого байта. Подмножествами могут быть буквы верхнего регистра, нижнего регистра, цифры и т. д. Для обращения к таблице используются два дескриптора: адрес таблицы и маска. В маске, представляющей собой один бит, единицами указывается, какие подмножества используются, причем одновременно это могут быть признаки нескольких из них. Так, например, используя объединение признаков приведенных подмножеств, можно указать все алфавитно-цифровые символы. Как и при адресации строки, обращение к таблице возможно при помощи двух форматов команд.

Символьные команды CIS-11

MOVC(I) — переслать строку символов,
MOVRC(I) — реверсировать и переслать строку символов,
MOVTC(I) — перекодировать строку символов по заранее определенной таблице и переслать,
CMPC(I) — сравнить две символьные строки,
MATC(I) — найти подстроку в строке,
LOCC(I) — найти первую позицию данного символа в строке,
SKPC(I) — пропустить все символы вплоть до символа, следующего за данным,
SCANC(I) — найти первую позицию символа, входящего в заданное подмножество,
SPANC(I) — найти первую позицию символа, не входящего в данное подмножество.

Десятичные операнды делятся на два класса: упакованные и неупакованные числа. Отличаются они тем, что в первом классе в байт упаковываются две цифры, а во втором — одна. Упакованные числа, в свою очередь, могут быть целым со знаком или абсолютным целым значением. Неупакованные также делятся на несколько подклассов, выделяемых в зависимости от того, чем заполняются неиспользуемые в байте разряды и где размещается знак числа. Если биты $4 \div 7$ во всех байтах равны 0, то такая форма называется «зоной со знаком» (или без знака). Если же эти разряды не нулевые, то возможны еще четыре подкласса десятичных чисел, соответствующих представлению в коде ASCII. Кроме того, существует еще одна дополнительная форма для упаковки целого числа (длинное целое) в двух 16-разрядных словах. Таким образом, число не может превышать $2^{31}-1$. Для адресации к десятичным операндам используются двухсловные дескрипторы, в которых указываются форма числа, его длина и адрес.

Над упакованными числами возможны следующие операции:

ADDP(I) — сложение,
SUBP(I) — вычитание,
MULP(I) — умножение,
DIVP(I) — деление с целым частным,
ASHP(I) — арифметический сдвиг,
CMPN(I) — сравнение.

Над неупакованными числами определены 4 операции:

ADDN(I) — сложение,
SUBN(I) — вычитание,
ASHN(I) — сдвиг,
CMPN(I) — сравнение.

Кроме того, имеется шесть команд преобразования:

CVTNL — перевод из неупакованной формы в длинное целое,
CVTLN — перевод из длинного целого в неупакованную форму,
CVTLP — перевод из длинного целого в упакованную форму,
CVTPL — перевод из упакованной формы в длинное целое,
CVTNP — перевод из неупакованной формы в упакованную,
CVTPN — перевод из упакованной формы в неупакованную.

Завершает набор команд CIS-11 команда загрузки дескрипторов в регистры. В зависимости от числа операндов возможны ее варианты:

L2Dr — загрузка двух дескрипторов,
L3Dr — загрузка трех дескрипторов.

Еще одним элементом современных моделей PDP-11 являются *средства защиты программного обеспечения*, появление которых вызвано мультипрограммным режимом работы ЭВМ. Основой этих средств является выделение нескольких режимов функционирования CPU. В моделях 11/44 и 11/70 их три — режим ядра (Kernel), супервизора (Supervisor) и пользователя (User). В режиме ядра возможно выполнение любой команды из системы команд, в то время как для всех остальных режимов команды HALT, RSET, SPL запрещены, попытка их выполнения приведет к прерыванию по вектору 4. Аналогичными ограничениями отличается режим пользователя от режима супервизора. Каждому из режимов соответствует свой собственный набор регистров диспетчера памяти и указатель стека. Кроме того, для увеличения скорости работы в режиме мультипрограммирования может быть удвоено число регистров R0 ÷ R6(11/70). Один из наборов используется при работе в режиме ядра и супервизора, а второй — при работе в режиме пользователя.

Д.2. СОВРЕМЕННЫЕ ВЫЧИСЛИТЕЛЬНЫЕ КОМПЛЕКСЫ НА БАЗЕ ЭВМ PDP-11

Д.2.1. Характеристики наиболее распространенных моделей PDP-11

11/04 и 11 34. Из выпускаемых ЭВМ наиболее близки к базисной, показанной на рис. 3.1, причем процессор 11/04 — это одноплатный вариант 11/34 с несколько уменьшенными возможностями. Общими для обеих машин являются: конструктивная база, пульт, программа первоначальной загрузки с автоматическим тестированием процессора. 11/04, единственная из выпускаемых моделей семейства, не имеет диспетчера памяти, поэтому размер оперативной памяти у нее ограничен 32 Ксловами. Кроме диспетчера памяти, преобразующего 16-разрядные виртуальные адреса в 18-разрядные физические и тем самым позволяющего расширить память до 256 Кбайт, 11/34 допускает использование таких дополнительных средств как аппаратная реализация арифметики с фиксированной (EIS) и плавающей точкой (FPP), а также кэш-памяти размером 2К. В целом эти средства повышают производительность 11/34 по сравнению с 11/04 примерно в 2,5 раза.

11/24. Выпуск в 1981 г. этой модели знаменует переход на более высокую ступень интегральной технологии. В 11/24 центральный процессор, диспетчер памяти и процессор с плавающей точкой реализованы на трех чипах.

Первый из них, называемый чипом данных (data chip), содержит регистры общего назначения (GPR), регистры слова состояния процессора (PSW), арифметическое и логическое устройство (ALU). В нем аппаратно реализованы также команды условного перехода. Набор регистров GPR имеет 2 регистра указателя стека R6(SP), что обеспечивает работу CPU в двух режимах: ядра и пользователя. Состав чипа предусматривает выполнение в нем всех арифметических и логических операций, пересылку данных и адресов в пределах, не требующих диспетчера памяти.

Второй чип — чип управления (control chip). Он включает логические схемы для выполнения микропрограмм, массивы микропрограмм (PLA) и массивы постоянной памяти (ROM). Повышение скорости выполнения микропрограмм достигается путем использования так называемого каскадного принципа (pipe-line), когда выполнение нескольких команд совмещено во времени. Для расширения функций количество чипов управления может быть увеличено, дополнительные должны быть подключены через чип выборки CSEL.

Наконец, чип управления памятью (MMU chip) выполняет преобразование 16-разрядного виртуального адреса задачи в 22- или 18-разрядный физический. Здесь же размещены 36 регистров, обеспечивающих работу FPP.

Архитектура 11/24 отличается от базисной дополнительной 22-разрядной шиной между CPU и памятью, минимизирующей время доступа процессора к памяти. Однако последняя остается по-прежнему подключенной к шине UNIBUS для обращения к ней устройств, подключенных к UNIBUS, в том числе внешней памяти на магнитных носителях.

11/44. Машина 11/44 по технологическому уровню близка к 11/24, а по своей производительности она соответствует более ранней модели 11/60. Основными особенностями 11/44 являются:

кэш-память размером 8 Кбайт;

отдельная 22-разрядная шина на тракте между процессором, кэш-памятью и оперативной памятью;

диспетчер памяти, допускающий ее расширение до 1 Мбайта;

дополнительные процессоры для реализации набора команд с плавающей точкой и набора команд CIS-11;

три режима работы CPU и соответственно утроенное число регистров управления памятью и R6 (SP);

встроенный микропроцессор, обеспечивающий ввод данных и программ в машинных кодах, регистрацию работы машины и возможность подключения ее к удаленной системе диагностики.

Новым элементом, появившимся в моделях 11/24 и 11/44, является также алфавитно-цифровой пульт процессора, заменивший собой традиционные клавишные регистры и индикационные лампочки. В качестве пульта используется системный терминал, например LA-120 или LA-36, который в таком случае имеет два режима. Первый — обычный, а второй — «режим пульта», работа в нем возможна на остановленном процессоре. Переход из первого режима во второй может быть выполнен либо программно, либо с терминала; обратный переход возможен только с терминала. В режиме пульта взаимодействие с остановленным CPU осуществляется с использованием элементов языка, аналогичного языкам отладочных систем.

11/70. Эта машина является самой производительной из ЭВМ системы PDP-11 Она выпускается с 1974 г., поэтому ряд новинок, появившихся позже, в ней отсутствует, например CIS-11 и алфавитно-цифровой пульт. Основное отличие от базисной архитектуры этой машины состоит в использовании 22-разрядной шины, объединяющей процессор с кэш-памятью, оперативной памятью и контроллером дисковых запоминающих устройств, причем высокоскоростные контроллеры входят в состав CPU. Развитые средства управления памятью допускают ее расширение до 4 Мбайт. В первых моделях 11/70 применялась память на ферритовых сердечниках, а в дальнейшем ее заменили платы на полупроводниковых элементах. Кэш-память имеет размер 2К и (как и в 11/44) является обязательным элементом архитектуры. Возможности функционирования в мультипрограммном режиме существенно расширены тремя режимами работы CPU и удвоенным количеством регистров GPR. В 11/70 может быть использован FPP, добавлены развитые средства обеспечения надежности функционирования.

Во всех моделях PDP-11 большое внимание уделяется обеспечению *надежности, готовности к работе и ремонтопригодности.* Совокупность аппаратных и программных средств, служащих для решения этой задачи, объединена под названием RAMP. В разных моделях такие средства могут иметь определенные отличия, но в целом они обеспечивают следующие функции:

программа первоначальной загрузки, зашиваемая в постоянную память, тестирует работу системы команд и оперативной памяти;

в памяти используются коды с обнаружением ошибок и контролем четности адресов;

ошибки, возникающие при обмене с кэш-памятью, памятью и дисками, контролируются аппаратными и программными средствами;

ведется журнал регистрации ошибок, используемый для оценки работы машины:

полупроводниковая память снабжена аккумуляторами автономного аварийного питания;

имеется возможность выполнения диагностических процедур средствами операционной системы, работающей на машине;

для обнаружения неисправностей включено значительное по объему тестовое программное обеспечение.

В целях сопоставления возможностей рассмотренных моделей ЭВМ PDP-11 целесообразно провести краткий сравнительный анализ их производительности на основе оценки времени выполнения отдельных команд. При этом должны учитываться несколько факторов.

Во-первых, команды могут быть одно- и двухоперандными или могут вообще не иметь операндов. Суммарное время выполнения двухоперандной команды складывается из времени обработки адресов источника и приемника и собственно времени выполнения команды. Для однооперандной команды время выполнения суммируется только со временем обработки адреса источника; наконец, для безоперандных команд отсутствуют оба компонента обработки адресов.

Во-вторых, время обработки адресов, в свою очередь, обусловлено режимом адресации, а для машин с кэш-памятью — еще и тем, находится ли в ней адресуемая ячейка или нет. В зависимости от режима адресации может потребоваться от 0 до 4 обращений к памяти. Цикл обращения к памяти за счет попадания или непопадания в кэш-память отличается в среднем в 2—2,5 раза. Среднее его значение для машины 11/70 обычно принимается равным 0,41 мкс. Если выбрать в качестве параметра сравнения производительности машин время выполнения операции сложения при наиболее и при наименее благоприятных условиях, то можно получить следующие соотношения (время в мкс):

Модель	Время выполнения команды		
	минимальное	максимальное	максимальное с попаданием в кэш-память
11/04	3,3	13,5	—
11/34	2,16	10,96	—
11/44	0,78	8,5	3,8
11/70	0,30	8,8	3,36

Для машины 11/70 среднее время выполнения операции сложения составляет 3—3,5 мкс.

Д.2.2. Номенклатура и технические характеристики периферийного оборудования

Устройства внешней памяти. Внешняя память PDP-11 включает магнитные диски различных типов, имеющие разный объем и быстродействие, магнитные ленты, ленты DECtape и кассетные ленты. Кроме устройств фирмы-производителя (Digital Equipment Corporation), в ЭВМ семейства PDP-11 могут использоваться запоминающие устройства других фирм. Однако ниже мы рассмотрим лишь устройства, поставляемые фирмой DEC.

Накопители на магнитных лентах

Модель	Тип	Объем памяти	Скорость обмена
TA11	Кассетная лента	90 Кбайт	560 байт/с
TC11/TU56	DECtape	295 Кбайт	10 Кбайт/с
TS03	Магнитная лента	5 Мбайт	10 Кбайт/с
TM11/TU1C	То же	20 Мбайт	36 Кбайт/с
TU16	»	32 Мбайт	72 Кбайт/с

Накопители на магнитных дисках

Модель	Тип	Объем памяти	Среднее время доступа	Скорость обмена
RX11/RX01	Флоппи-диски	125 Кслов	483 мс	36 мкс/слово
RK11/RK05	Дисковая кассета	1,2 Мслов	70 мс	11 мкс/слово
RP11/RP03	Дисковый пакет	20 Мслов	42 мс	7 мкс/слово
RS03	Диск с фиксированными головками	256 Кслов	8,5 мс	4 или 8 мкс/слово
RS04	Диск с фиксированными головками	512 Кслов	8,5 мс	4 мкс/слово
RP04	Дисковый пакет	44 Мслов	36 мс	2,5 мкс/слово

Терминальные устройства. Здесь представляется целесообразным упомянуть о печатающих терминалах, видеотерминалах и, наконец, об интеллектуальных терминальных комплексах.

Печатающие терминалы наиболее широко представлены устройствами LA34, LA36 и LA120, имеющими скорость печати от 30 до 180 симв/с, число символов в строке до 216, скорость обмена с ЭВМ до 9600 бит/с, объемы буфера для обмена от 16 до 4096 символов. Эти терминалы включают интерфейсы для выхода на стандартный модемный интерфейс EIA или на 20 мА токовую петлю.

В качестве наиболее характерных *видеотерминалов* можно привести графический дисплей VT55, дисплеи VT62 и VT100. Такие терминальные устройства обеспечивают широкие возможности отображения различной информации и организации интерактивных взаимодействий с программами.

Интеллектуальные терминалы семейства PDT-11 включают микроЭВМ LSI-11, обеспечивающую возможность реализации ряда функций на терминальном уровне. Число и характер таких функций могут быть изменены путем написания соответствующих программ для LSI-11. Функционирование PDT-11 обеспечивается при помощи загрузчиков, размещенных в RAM и ROM памяти (память произвольного доступа и память только для чтения) и встроенных диагностических процедур. Устройства семейства PDT-11 имеют также набор последовательных интерфейсов для связи с ЭВМ (одной или несколькими) и устройствами печати LA180.

Устройства комплексирования и коммуникационные устройства. Реализация взаимодействия нескольких ЭВМ PDP-11 в составе многомашинных комплексов и сетей осуществляется с использованием специальных устройств.

Для комплексирования нескольких близко расположенных машин применяются переключатель шины DT03, адаптер связи шин DA11-B и устройство «окна» шины DA11-F. Построение территориально распределенных систем требует использования синхронных и асинхронных коммуникационных адаптеров и мультиплексоров, таких, как DH11, DL11, DUP11, DMC11, и аналогичных. Эти устройства применяются и при организации гетерогенных комплексов, содержащих, наряду с машинами семейства PDP-11, ЭВМ других фирм-изготовителей (IBM, HP, Burroughs и др.).

Завершая краткий обзор аппаратных средств системы PDP-11, отметим, что процессоры машин этого семейства могут комплектоваться целым рядом дополнительных устройств, например специализированными процессорами для выполнения матричных операций, преобразователями «вектор-растр» для использования электрографических графопостроителей и многими другими.

Д.3. ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ PDP-11

Программное обеспечение ЭВМ семейства PDP-11, поставляемое фирмой DEC, включает операционные системы RT-11, RSTS/E, RSX-11M/M-PLUS, RSX-11S, IAS, DSM-11, TRAX; программное обеспечение для организации сетей ЭВМ DECNET; программные средства для создания баз данных SORT-11,

RMS-11, DBMS и DATATRIEVE; Ассемблер MACRO-11, компиляторы с языков BASIC, Кобол, Фортран, APL и др. Помимо этих средств, широкое распространение получает программное обеспечение, создаваемое другими производителями, например операционная система UNIX, а также разработки пользователей машин фирмы DEC — DECUS. Здесь мы дадим краткую характеристику программного обеспечения, входящего в состав стандартных поставок.

Д.3.1. Операционные системы

Операционная *система реального времени RT-11* обладает наименьшими возможностями по сравнению с другими упомянутыми системами, но реализует их за счет минимальных затрат ресурсов ЭВМ. RT-11 может выполняться на всех моделях PDP-11 вплоть до 11/70 и реализуется в трех вариантах:

SJ — однопрограммная,
FB — однопрограммная с решением фоновых задач в пакетном режиме,
XM — аналогична FB, но имеет расширенные функции управления памятью.

Два первых варианта RT-11 допускают функционирование на машинах с размером оперативной памяти не более 32, последний — до 128 Кслов.

Взаимодействие пользователя с системой осуществляется посредством языка монитора, а программ — при помощи набора системных директив. Разработка новых средств программного обеспечения существенно облегчается применением редактора текстов RT-11 EDIT и вспомогательных утилит. Система RT-11 включает отладочные средства и системную библиотеку SYSF4, обращение к которой возможно из программ, написанных на Фортране. Средства библиотеки позволяют осуществить взаимодействие между задачами переднего и фонового разделов, синхронизировать I/O операции, обрабатывать символьные строки и др.

В рамках RT-11, помимо Фортрана, можно использовать языки MACRO-11, APL, BASIC, FOCAL. RT-11 не ориентирована на работу с базами данных.

Операционная *система разделения времени и ресурсов* RSTS/E — это многопользовательская система, обслуживающая до 64 пользователей. Кроме режима интерактивных взаимодействий пользователей со своими программами, в ней возможна также организация пакетного режима выполнения заданий. В основном RSTS/E предназначена для пользователей, работающих на языке BASIC-PLUS, но в ней допускается использование и таких языков, как Кобол, Фортран, BASIC-PLUS2, APL. RSTS/E может функционировать на всех моделях, кроме 11/04. Система RSTS/E включает три основных элемента:

1. Системный код, который, в свою очередь, состоит из векторов прерываний; системных буферов и таблиц; драйверов устройств; файлового процессора.

2. Языковые процессоры. По умолчании используется языковый процессор BASIC-PLUS. Он обеспечивает создание текстов программ (при помощи собственного редактора текстов), их компиляцию и выполнение. Элементы языкового процессора, выполняющие эти функции, резидентны в памяти. Временно могут загружаться процессоры других языков с соответствующими средствами для разработки и выполнения программ.

3. Библиотека служебных программ для обслуживания системы, учета и распределения ресурсов, оперативного управления системой, регистрации и анализа ошибок, организации взаимодействия как между пользователями и оператором системы, так и между несколькими пользователями. Интерфейс общения с системой основывается на относительно простом языке управления CCL, включающем около 40 команд, а для организации пакетного режима применяется специальный язык управления заданиями.

Являющиеся основными в RSTS/E языки BASIC-PLUS и BASIC-PLUS2 снабжены широким набором служебных функций, которые, в частности, позволяют управлять выполнением программ, получать учетную информацию, назначать устройства, осуществлять все необходимые действия по обслуживанию каталогов файлов на дисках и лентах.

RSTS/E не предназначена для работы в режиме реального времени.

Семейство операционных *систем реального времени RSX-11* представлено системами RSX-11M, RSX-11M/PLUS и RSX-11S. Эти операционные системы имеют широкие возможности и могут быть использованы для значительной части областей применения машин PDP-11. RSX-11M — компактная многопользовательская система, допускающая выполнение на любой из моделей PDP-11. Она обеспечивает возможность интерактивной работы пользователей с терминалов со своими программами одновременно с выполнением задач в режиме реального времени. Система состоит из управляющей программы, монитора взаимодействия пользователя с системой (MCR), построителя задач (TKB) и набора утилит. В RSX-11M могут быть включены трансляторы с языков MACRO-11, Кобол, BASIC-11, BASIC-PLUS2, Фортран IV и Фортран IV-PLUS (одна из версий языка Фортран 77).

Система RSX-11M/PLUS предназначена для выполнения на старших моделях PDP-11. Она лучше остальных систем RSX-11 использует возможности мощных моделей PDP-11 посредством оперирования тремя состояниями: пользователя, супервизора и ядра. Эта операционная система имеет более развитые средства защиты программ и данных в многопользовательском режиме, а также позволяет организовывать выполнение задач в пакетном режиме. Наконец, система RSX-11S представляет собой подмножество RSX-11M, специализированное исключительно для работы в системах управления в режиме реального времени. RSX-11S резидентна в оперативной памяти, из нее исключены средства для разработки нового программного обеспечения.

Операционные системы из семейства RSX-11 имеют эффективные средства динамического распределения оперативной памяти, развитую файловую систему File-11 и могут быть использованы совместно с различными базами данных.

Интерактивная операционная система IAS — это самая мощная из универсальных операционных систем, работающих на PDP-11. Она допускает использование на всех выпускаемых ЭВМ семейства, кроме 11/04 и 11/44.

IAS имеет две управляющие программы, одна из которых служит для управления задачами в режиме реального времени, а вторая — в режиме разделения времени. Время, не использованное задачами первой группы, передается для распределения между задачами второй группы. Распределение выполняется путем квантования времени на интервалы, пропорциональные приоритету задачи. Оставшееся время используется для задач, выполняемых в пакетном режиме.

IAS обладает более развитыми, чем RSX-11, средствами защиты программ и данных в многопользовательском режиме. Если в RSX-11 любой из терминалов подключается к монитору системы MCR, получая при этом доступ практически ко всем ресурсам системы, то в IAS, где аналогом MCR служит система PSD, для отдельного терминала этот доступ может быть более ограниченным, например допускать взаимодействия только с программой-редактором.

IAS поддерживает тот же набор языков, что и RSX-11.

Многопользовательская операционная система DSM-11 в основном ориентирована на обработку текстовой информации с использованием специализированного языка высокого уровня MUMS. Ядром системы является языковый процессор интерактивного типа, включающий в себя все средства, необходимые для подготовки текстов программ, их трансляции и выполнения. Кроме языкового процессора, в состав DSM-11 входят средства организации базы данных, обеспечивающие хранение любого вида текстовых данных в виде иерархически упорядоченного набора глобальных переменных и массивов. Система DSM-11 автоматически отображает эти переменные и массивы на диск в форме блоков записей фиксированного размера. Таким образом, исключается обычный способ обращения к внешней памяти с указанием устройства и имени файла. С целью увеличения скорости обмена с диском в оперативной памяти выделяется специальный буфер, выполняющий функции, аналогичные выполняемым кэш-памятью.

В зависимости от размера ОЗУ система допускает подключение до 80 пользователей, из которых 63 могут работать одновременно. Каждому пользователю выделяется раздел памяти фиксированного размера (до 4 Кбайт). По возможности доступа к системе пользователи делятся на чисто эксплуатационный

персонал, осуществляющий работу с базой данных системы, и программистов, которым поручается разработка новых программ.

В DSM-11 большое внимание уделяется сохранности данных, для чего, в частности, ведется на магнитной ленте журнал, в котором фиксируются все операции обмена с диском. В случае аварии диска по содержимому этого журнала диск может быть восстановлен. Для ведения журнала используются поочередно два буфера с тем, чтобы обеспечить синхронизацию работы системы со скоростью записи на магнитную ленту.

DSM-11 представляет собой систему закрытого типа, т. е. в ней не могут быть применены какие-либо нестандартные средства организации баз данных или отличные от MUMS языки.

Специализированная система коммерческого применения TRAX служит для работы с большими объемами информации и передачи сообщений. В рамках TRAX может быть использован терминал только типа VT-62. Он приспособлен для ввода и вывода данных в форме заранее подготовленных бланков и позволяет оптимизировать передачу данных, хранение бланков и проверку данных при заполнении бланков.

TRAX состоит из двух основных составных частей: процессора передачи сообщений (transaction processor) и вспомогательных средств. Процессор передачи сообщений включает программы и таблицы данных, обеспечивающие работу прикладных терминалов. Вспомогательные средства обеспечивают выполнение программ в пакетном режиме и работу вспомогательных терминалов в режиме разработки программного обеспечения и управления ресурсами системы.

Программирование прикладных терминалов выполняется на специализированном языке — языке прикладного терминала ATL. Этот язык позволяет описывать бланки и правила работы с ними, задавать режимы применения функциональных клавишей VT-62 и методы преобразования бланков в сообщения системе. Программирование на ATL выполняется с использованием стандартного редактора текстов EDT. Область применения TRAX — это банковские, торговые системы, системы регистрации изобретений и т. д.

Структура TRAX дает возможность подключения к более мощным системам аналогичного применения, базирующимся на машинах IBM/360 и IBM/370. Дополнительно для создания прикладного программного обеспечения TRAX допускает работу на языках Кобол и BASIC-PLUS2.

Д.3.2. Программное обеспечение систем телеобработки

Учитывая значительную и постоянно растущую потребность в средствах телеобработки, фирма DEC предлагает целый набор соответствующих пакетов. Среди них в первую очередь стоит отметить следующее.

Семейство пакетов DECNET, служащих для организации сетей ЭВМ на базе PDP-11, представляет собой набор средств, реализующих сетевую архитектуру DNA (Digital Network Architecture) в рамках операционных систем RT-11, RSTS/E, RSX-11 и IAS.

DECNET обеспечивает построение сетей, имеющих различную топологию и географическую распределенность, и выполняет следующие основные функции:

безошибочную передачу информации между разными узлами сети;

транспортировку сообщений по сети;

установление, поддержание и уничтожение логических каналов между различными программами сети;

дистанционную пакетную обработку;

обмен файлами между узлами сети;

дистанционное интерактивное взаимодействие пользователя с его программой;

взаимодействия между программами разных узлов сети;

терминальные взаимодействия;

контроль и тестирование сети и сбор статистики о ее функционировании.

Использование DECNET дает возможность объединять в сети различные модели PDP-11, функционирующие под управлением разных операционных си-

стем. Использование средств DECNET позволяет получить доступ к сети программ, написанных на языках MACRO-11, Фортран, Кобол и BASIC.

Пакет PE3271 служит для эмуляции группового терминального устройства IBM 3271 на PDP-11. Это обеспечивает возможность организации следующих взаимодействий между PDP-11 и машинами фирмы IBM:

обмен файлами,
доступ с терминала PDP-11 к файлам в IBM,
управление программами в IBM с терминалов PDP-11,
взаимодействие программ, выполняемых в различных ЭВМ.

Пакет PE3271 поддерживает коммуникации, реализуемые с использованием выделенных или коммутируемых линий связи либо физических линий.

Пакет SNA/PE служит для включения ЭВМ PDR-11 в сеть машин IBM, связанных в соответствии с архитектурой SNA (System Network Architecture). В SNA/PE реализуется эмуляция группового коммуникационного контроллера IBM 3790. Основное назначение этого пакета состоит в предоставлении возможностей создания неоднородных сетей ЭВМ на базе машин PDP-11 и IBM. При этом программы, выполняемые в PDP-11, могут пользоваться всеми услугами сети и, следовательно, всеми ресурсами мощных ЭВМ моделей IBM.

Д.3.3. Программное обеспечение баз данных

Для организации баз данных и доступа к ним фирма DEC поставляет следующее программное обеспечение.

Система управления данными RMS-11 — это универсальная система, расширяющая возможности программирования файловых операций и облегчающая написание и отладку соответствующих программ. RMS-11 может быть включена в системы IAS, RSX-11 и RSTS/E в процессе генерации. Применение RMS-11 дает возможность создавать файлы с последовательной, произвольной и индексной организацией. Для доступа к ним могут быть использованы последовательный и произвольный методы, а также работа непосредственно по адресу записи. Кроме трех основных методов имеется динамический метод доступа, обеспечивающий наиболее эффективную работу с файлами за счет переключения с одного из основных на другой. Например, первая запись читается с прямым доступом, а все последующие — последовательным доступом.

RMS-11 включает значительный набор средств для оптимизации работы с файлами, предоставляет возможность работы нескольких пользователей с одним файлом одновременно и имеет развитую систему буферизации и защиты.

В системах IAS, RSX-11 и RSTS/E для доступа к файлам, организованным при помощи RMS-11, может быть использована *система DATATRIEVE.* Она представляет собой относительно простое средство для интерактивного доступа к данным, их модернизации и сортировки, генерации отчетов и т. д. DATATRIEVE имеет развитый входной язык, предложения которого интерпретируются по отдельности, или с применением операторов управления из этих предложений можно создавать небольшие программы, обеспечивающие требуемые манипуляции с данными.

SORT-11 — вспомогательные программы для работы с файлами RMS-11. С их помощью выполняется процесс сортировки, в результате которого из входного файла формируется новый, выходной. Целью сортировки может быть переупорядочение записей файла, выборка из файла определенных записей и др. Выходной файл может состоять непосредственно из отсортированных записей или представлять собой файл ключей или адресов отсортированных записей.

Для *организации баз данных в соответствии со стандартами CODASYL служит система DBMS.* В ее состав входят методы организации данных, языки описания данных и оперирование ими, описания схемы данных, методы их физической и логической организации, а также вспомогательные программы для работы с ними. Непосредственное взаимодействие с DBMS осуществляется на языке оперирования данными DBMS-11, кроме того, в программах, написанных на языках Фортран, Кобол, BASIC и MACRO-11, могут быть использованы средства языка DML, обеспечивающего для прикладных программ доступ к базе данных.

ЛИТЕРАТУРА

Abrams M. D. and Stein P. G. Computer Hardware and Software: An Interdisciplinary Introduction. Addison — Wesley, Reading, Massachusetts, 1973.

Aho A. V., Hopcroft J. E. and Ullman J. D. The Design and Analysis of Computer Algorithms, Addison — Wesley, Massachusetts, 1976.

Baase S. Computer Algorithms: Introduction to Design and Analysis, Addison — Wesley, Massachusetts, 1978.

Barron D. W. Recursive Techniques in Programming. American Elsevier, New York, 1968. Русский перевод: Баррон Д. Рекурсивные методы в программировании. М., Мир, 1974.

Bell C. G. and A. Newell. Computer Structures: Readings and Examples. McGraw-Hill. New York, 1971.

Bell C. G., Mudge J. C. and McNamara J. Computer Engineering. Digital Press, Maynard, Ma., 1978.

Bell J. Threaded code. — Communications of the ACM, vol. 16, June 1973, p. 370—372.

Berztiss A. T. Data Structures — Theory and Practice. Academic Press. New York, 1971.

Blankinship W. A. Note on computing autocorrelations. — IEEE Trans. Acoustics, Speech, Signal Processing, vol. ASSP-22, February 1974, p. 76—77.

Brender R. F. Turning cousins into sisters: an example of software smoothing of hardware differences. — In: Computer Engineering, ibid Bell et. al.

Brillinger P. C. and D. J. Cohen. Introduction to Data Structures and Nonnumeric Computations. Prentice-Hall, Englewood Cliffs, N. J. 1972.

Brinch Hansen P. Operating System Principles. Prentice-Hall, Englewood Cliffs. N. J., 1973.

Broihier J. C. and Crowley M. J. A RSTS/E audio response application. — DECUS Proceedings. Boston, May 1977, p. 1355—1375.

Burr W. E. and Smith W. R. Comparing computer architectures. — Datamation, vol. 23, February 1977, p. 48—52.

Chamberlin H. A sampling of techniques for computer performance of music. — BYTE, vol. 2, September 1977, p. 62—83.

Cohen L. J. Operating Systems Analysis and Design. Spartan Books, New York, 1970. Русский перевод: Коэн Л. Дж. Анализ и разработка операционных систем. М., Наука, 1975.

Denning P. J. Third Generation Computer Systems. — Computing Surveys, vol. 3, No 4, 1971.

Donovan J. J. Systems Programming. McGraw-Hill, New York, 1972. Русский перевод: Донован Дж. Системное программирование. М., Мир, 1975.

Elovitz H. S., Johnson R. W., McHugh A. and Shore J. E. Automatic translation of English text to phonetics by means of letter-to-sound rules. — NRL Report 7948, January 21, 1976.

Flanagan J. L. Speech Analysis. Synthesis and Perception, Springer-Verlag, New York, 1972.

Flanagan J. L. and Rabiner L. R. Speech Synthesis Dowden, Hutchison and Ross, Pennsylvania, 1973.

Flores I. Computer Organization. Prentice-Hall, Englewood Cliffs. N J., 1969. Русский перевод: Флорес И. Организация вычислительных машин. М., Мир, 1972.

Flores I. Data Structure and Management. Prentice-Hall, Englewood Cliffs. N. J., 1970. Русский перевод: Флорес И. Структуры и управление данными. М., Финансы и статистика, 1982.

Foster C. C. Computer Architecture. Van Nostrand Reinhold. New York, 1970.

Gagnon R. T. VOTRAX real time hardware for phoneme synthesis of speech. — Conf. Rec., 1978 IEEE Iiternational Conf. on Acoustics, Speech and Signal Processing. Tulsa, Oklahoma, p. 175—178.

Gear C. W. Computer Organization and Programming. 2nd ed., McGraw-Hill, New York, 1974.

Giloi W. K. Interactive Computer Graphics: data structures, algorithms, languages. Prentice-Hall, Englewood Cliffs. N. J., 1978.

Haberman A. N. Introduction to Operating Systems Design. Science Research Associates. Chicago, 1976.

Hayes J. P. Computer Architecture and Organization. McGraw-Hill. New York, 1978.

Hellerman H. Digital Computer System Principles. 2nd ed., McGraw-Hill. New York, 1973.

Holt R. C. Structured Concurrent Programming with Operating Systems. Addison — Wesley, Reading, Massachusetts, 1978.

Husson S. S. Microprogramming: Principles and Practices. McGraw-Hill. New York, 1970.

Jackson T. L. PDP-11/60 Microprogramming Specification. Digital Equipment Corporation, Massachusetts, 1977.

Johnson L. R. System Structure in Data, Programs and Computers. Prentice-Hall, Englewood Cliffs. N. J., 1970.

Katzan H. Computer Organization and the System/370. Van Nostrand Reinhold. New York, 1971. Русский перевод: Катцан Г. Вычислительные машины системы 370. М., Мир, 1974.

Katzan H. Jr. Operating Systems — A Pragmatic Approach. Van Nostrand Reinhold. New York, 1973. Русский перевод: Катцан Г. Операционные системы. М., Мир, 1976.

Kent W. Assembler-Language Macroprogramming. — Computing Surveys, vol. 1, No 4, 1969.

Knuth D. E. The Art of Computer Programming, vol. 1: Fundamental Algorithms, Addison — Wesley, Reading, Massachusetts, 1968. Русский перевод: Кнут Д. Искусство программирования для ЭВМ. Т. I. Основные алгоритмы. М., Мир, 1976.

Lewis T. G. and Smith M. Z. Applying Data Structures. Houghton Mifflin. Boston, 1976.

Lowry E. S. and Medlock C. W. Object code optimization. — Communications of the ACM, vol. 12, Jan. 1969, p. 13—22.

Markel J. D. and Gray A. H. Linear Prediction of Speech. Springer-Verlag. New York, 1976.

Mauer W. D. Programming: An Introduction to Computer Languages and Techniques. Holden-Day, San Francisco, 1968.

Maurer H. H. Data Structures and Programming Techniques. Prentice-Hall, Englewood Cliffs. N. J., 1977.

McNamara J. E. Technical Aspects of Data Communication. Digital Press, Massachusetts, 1977.

Moorer J. A. Signal processing aspects of computer music. — Proceedings of the IEEE, vol. 65, August 1977, p. 1108—1137.

Morris L R. Fast speech spectrogram reduction and display on minicomputer/graphics processors. — IEEE Trans. Acoustics, Speech, Signal Processing, vol. ASSP-23, June 1975, p. 297—300.

Morris L. R. Fast digital signal processing software package for the PDP-11.— DECUS Program Library, No 11—296, Nov. 1976.

Morris L. R. Automatic generation of time efficient digital signal processing software. — IEEE Trans. Acoustics, Speech, Signal Processing, vol. ASSP-25, February 1977, p. 74—79.

Morris L. R. and Mudge J. C. Speed enhancement of digital signal processing software via microprogramming a general purpose computer. — IEEE Trans. Acoustics, Speech, Signal Processing, vol. ASSP-26, April 1978, p. 135—139.

Morris L. R. A comparative study of time efficient FFT and WFTA programs for general purpose computers. — IEEE Trans. Acoustics, Speech, Signal Processing, vol. ASSP-26, April 1978, p. 141—150.

Morris L. R. and Pearlman M. A software package for fast speech spectrogram generation and display on PDP-11 computer/graphics processors. — DECUS Program Library, № 11—361, July 1978.

Morris L. R. Fast FORTRAN/RT-11 Syslib software for implementing the Naval Research Laboratory «English text to VOTRAX parameters» algorithm. — DECUS Program Library, № 11—375, November 1978.

Morris L. R. and Allan D. I. An LPC k-parameter software speech synthesizer via dynamic microprogramming a general purpose computer.—Conf. Record, IEEE International Conference on Acoustics, Speech and Signal Processing Washington, D. C., April 1979.

Mudge J. C. Design decisions achieve price/performance balance in mid-range computers. — Computer Design, August 1977, p. 87—95.

Newman W. and Sproull R. Principles of Interactive Computer Graphics. McGraw-Hill. New York, 1973.

Oppenheim A. V. and Schafer R. W. Digital Signal Processing, Prentice-Hall, Englewood Cliffs. N. J., 1975.

Organick E. I. The Multics System: An Examination of its Structure. MIT Press, Cambridge, Mass., 1972.

Pfeifer L. L. Multiplication reduction in short-term autocorrelation. — IEEE Trans. Audio and Electroacoustics, vol. AU-21, December 1973, p. 556—558.

Phister M. Logical Design of Digital Computers. Wiley. New York, 1959.

Presser L. and White J. R. Linkers and Loaders. — Computing Surveys, vol. 4, № 3, 1972.

Rabiner L. R. and Gold B. Theory and Application of Digital Signal Processing. Prentice-Hall, Englewood Cliffs. N. J., 1975.

Rabiner L. R. and Schafer R. W. Digital Processing of Speech Signals. Prentice-Hall, Englewood Cliffs. N. J., 1978.

Ralston A. Introduction to Programming and Computer Science. McGraw-Hill. New York, 1971.

Rosin R. F. Contemporary Concepts of Microprogramming and Emulation. Computing Surveys, vol. 1, № 4, 1969.

Rubinfeld L. P. A proof of the modified Booth's algorithm for multiplication. — IEEE Trans. on Computers, vol. C-24, October 1975, p. 1014—1015.

Shaw A. C. Logical Design of Operating Systems. Prentice-Hall, Englewood Cliffs. N. J., 1974. Русский перевод: Шоу А. Логическое проектирование операционных систем. М., Мир, 1981.

Sheingold D. H. Analog-digital Conversion Handbook, Analog Devices, Norwood, Massachusetts.

Singleton R. C. An algorithm for computing the mixed radix fast Fourier transform. — IEEE Trans. Audio and Electroacoustics, vol. AU-17, June 1969, p. 93—103.

Snyder D. C. Computer performance improvement by measurement and microprogramming, Hewlett — Packard Journal, February 1975, p. 17—24.

Steiglitz K. An Introduction to Discrete Systems, John Wiley. New York, 1974.

Stone H. S. and Siewiork D. Introduction to Computer Organization and Data Structures. McGraw-Hill. New York, 1975.

Stone H. S. Introduction to Computer Architecture. SRA, Chicago, 1975.

Strecker W. D. Cache memories for PDP-11 family computers. — Proc. 3rd Annual Symposium on Computer Architecture, 1976, p. 155—158.

Szymanski T. G. Assembling code for machines with span dependent instructions. — Comm. ACM, vol. 21, April 1978, p. 300—308.

Tannenbaum A. S. Structured Computer Organization. Prentice-Hall, Englewood Cliffs. N. J., 1976. Русский перевод: Таненбаум Э. Многоуровневая организация ЭВМ. М., Мир, 1979.

T-Thong and Liu B. Fixed-point fast Fourier transform error analysis, — IEEE Trans. Acoustics, Speech, Signal Processing, vol. ASSP-24, December 1976, p. 563—573.

Van Dam A. and Rice D. E. On-Line Text Editing: A Survey. — Computing Surveys, vol. 3. № 3, 1971.

Wald B. et al. Military Computer Architectures. Special issue of Computer. IEEE, October 1977.

Walker T. M. Introduction to Computer Science: An Interdisciplinary Approach. Allyn and Bacon, Boston, 1972.

Watson R. W. Timesharing System Design Concepts. McGraw-Hill. New York, 1970.

Wegner P. Programming Languages, Information Structures and Machine Organization, McGraw-Hill. New York, 1968.

Wiggins R. and Brantingham L. Three-chip system synthesizes human speech. — Electronics, vol. 51, August 31, 1978, p. 109—116.

Wirth N. Algorithms + Data Structures = Programs. Prentice-Hall, Englewood Cliffs. N. J., 1976.

PDP-11 Peripherals Handbook. Digital Equipment Corp., Maynard, Mass., 1978.

PDP-11/04/34/45/55/60 Processor Handbook. Digital Equipment Corp., Maynard, Mass., 1978.

ОГЛАВЛЕНИЕ

Предисловие к русскому переводу 5
Предисловие 7

Глава 1. **Введение** 13
1.1. Чего требует ЭВМ? 13
1.2. Области применения ЭВМ 14
1.3. Возможности и ограничения вычислительной машины 16
1.4. Архитектура, организация и реализация ЭВМ 16

Глава 2. **Технические средства и программное обеспечение ЭВМ** . . . 19
2.1. Основные элементы технических средств 19
2.2. Программное обеспечение ЭВМ. Введение в программирование . . . 23
2.3. Архитектура адресации операндов 30
2.4. Описание и обработка адресов 36
2.5. Этапы выполнения команд 45
2.6. Символический Ассемблер, средства программирования 48
Упражнения 53
Библиография 54

Глава 3. **ЭВМ PDP-11** 55
3.1. Структура машины PDP-11 56
3.2. Представление данных 61
3.3. Форматы команд и режимы адресации PDP-11 68
3.4. Ассемблер MACRO-11 86
3.5. Семейство PDP-11 и иерархия памяти 89
Упражнения 98
Библиография 100

Глава 4. **Структуры программ и методы программирования** 101
4.1. Команда перехода 101
4.2. Подпрограммы 102
4.3. Позиционно-независимое программирование 110
4.4. Развитие концепции подпрограмм 116
4.5. Макрокоманды и макроассемблеры 120
Упражнения 131
Библиография 134

Глава 5. **Структуры данных и работа с ними** 135
5.1. Обработка байт и коды символов 135
5.2. Арифметические операции с фиксированной точкой 136
5.3. Логические операции 152
5.4. Арифметика с плавающей точкой 153
5.5. Внутреннее и внешнее представление данных 160
5.6. Введение в структуры данных 162
Упражнения 186
Библиография 186

Глава 6. **Микромашинная архитектура и микропрограммирование для PDP-11** 187
6.1. Введение в микропрограммные машины 187
6.2. Магистрали данных PDP-11/60 189
6.3. Микроциклы 195
6.4. Дерево сдвига и регистр сдвига: структура и использование 197
6.5. Магистраль ввода-вывода 200
6.6. Концепция остаточного управления 200
6.7. Последовательности микрокоманд и микротесты переходов 201

6.8. Параллелизм в микрокодировании: команда MUL 204
6.9. Заключение по главе 205
Упражнения 206
Библиография 206

Глава 7. **Программирование ввода-вывода** 207
7.1. Основы программирования операций ввода-вывода и их выполнение 208
7.2. Базовые устройства 209
7.3. Проблема начальной загрузки 218
7.4. Устройства памяти на лентах и дисках 223
7.5. Прерывания 233
7.6. Записи и буферизация 247
7.7. Системы программирования ввода-вывода (IOPS) 248
7.8. Диспетчер памяти и PSW 251
Упражнения 258
Библиография 258

Глава 8. **Системные средства разработки программного обеспечения** 259
8.1. Редактор 259
8.2. Загрузчик 264
8.3. Методы отладки 270
8.4. Прикладное программное обеспечение и его вычислительная среда . 273
Упражнения 274
Библиография 274

Глава 9. **Интерактивные вычисления в реальном времени** 275
9.1. Интерактивные графические процессоры 275
9.2. Взаимодействие CPU, DPU и A/D при обработке цифрового сигнала 285
Библиография 296

Глава 10. **Операционные системы** 297
10.1. Простейшая вычислительная система 297
10.2. Компоненты дисковой операционной системы 298
10.3. Использование операционных систем 313
10.4. Системы, работающие в пакетном режиме и режиме разделения времени 314
10.5. Системы управления, работающие в режиме реального времени . 316
10.6. Системы баз данных 317
10.7. Коммуникационные вычислительные системы 319
Библиография 321

Глава 11. **Структура и принципы функционирования мультипрограммных операционных систем** 322
11.1. Общий обзор MMS 322
11.2. Структура очередей, таблиц и PSECT 325
11.3. Супервизор MMS 328
11.4. Написание программ для выполнения под управлением MMS . . 332
11.5. Расширенная система 333
11.6. Листинг MMS 334

Упражнения 339
Приложение 1. **Пульт оператора** 340
П.1. Составные части пульта 340
П.2. Работа управляющих переключателей 342

Приложение 2. **Дополнение к русскому изданию: обзор моделей ЭВМ семейства PDP-11 и их программного обеспечения** . . . 344
Д.1 Новые элементы архитектуры 344
Д.2 Современные вычислительные комплексы на базе ЭВМ PDP-11 . . . 346
Д.3. Программное обеспечение PDP-11 349

Р. Экхауз, Л. Моррис

Мини-ЭВМ: Организация и программирование

Книга одобрена на заседании секции редсовета по электронной обработке данных в экономике 18.06.1981 г.

Зав. редакцией *А. В. Павлюков*
Редактор *К. М. Чижевская*
Мл. редакторы *О. А. Ермилина и О. Б. Степанченко*
Техн. редактор *Г. А. Полякова*
Корректоры *Г. А. Башарина, Я. Б. Островский, Т. М. Иванова, В. Б. Голяндичева*
Худож. редактор *О. Н. Поленова*
Переплет художника *Н. С. Филиппова*
ИБ № 1285

Сдано в набор 16.04.83. Подписано в печать 02.09.83. Формат 60×90¹/₁₆. Бум. тип № 2. Гарнитура «Литературная». Печать высокая. Усл. п. л. 22,5. Уч.-изд л. 24,09. Тираж 25 000 экз. Заказ 2131. Цена 1 р. 80 к.

Издательство «Финансы и статистика», 101000, Москва, ул. Чернышевского, 7

Отпечатано с матриц Ленинградской типографии № 2 головного предприятия ордена Трудового Красного Знамени Ленинградского объединения «Техническая книга» им. Евгении Соколовой Союзполиграфпрома при Государственном комитете СССР по делам издательств, полиграфии и книжной торговли. 198052, г. Ленинград, Л-52, Измайловский проспект, 29, в Ленинградской типографии № 4 ордена Трудового Красного Знамени Ленинградского объединения Техническая книга» им. Евгении Соколовой Союзполиграфпрома при Государственном комитете СССР по делам издательств, полиграфии и книжной торговли. 191126, Ленинград, Социалистическая ул., 14.